Pontifícia Universidade Católica
do Rio de Janeiro

Eiji Adachi Medeiros Barbosa

Global-Aware Recommendations for Repairing Exception Handling Violations

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC–Rio as partial fulfillment of the requirements for the degree of Doutor em Informática

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
November 2015

Eiji Adachi Medeiros Barbosa

Global-Aware Recommendations for Repairing Exception Handling Violations

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC–Rio as partial fulfillment of the requirements for the degree of Doutor em Informática.

Rio de Janeiro — November 30th, 2015

Eiji Adachi Medeiros Barbosa

Eiji Adachi Medeiros Barbosa received his Bachelor degree in Computer Science from the Federal University of Rio Grande do Norte (UFRN) in 2009. He received his Master degree in Informatics from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2012. His main research interests are Software Engineering, Exception Handling and Recommender Systems for Software Engineering.

Bibliographic data

Barbosa, Eiji Adachi Medeiros

Global-Aware Recommendations for Repairing Exception Handling Violations / Eiji Adachi Medeiros Barbosa; Advisor: Alessandro Fabricio Garcia. — 2015.

v., 213 f: il. ; 29,7 cm

1. Tese (Doutorado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia.

1. Informática – Teses. 2. Tratamento de exceções. 3. Sistemas de recomendação para engenharia de software. 4. Políticas de tratamento de exceções. 5. Reparação de violações de tratamento de exceções. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my beloved grandmother,
Nobu Adachi (*in memoriam*).

Acknowledgments

I would like to express my deepest gratitude to my beloved wife Maria Luiza for being by my side from the first day to the last day of this journey. Your love, affection and support were my driving force to achieve this goal.

Although the proper words fail me to express my gratitude, I thank my parents, my sisters and all family for all the love, trust and encouragement. You are my greatest inspiration and motivation. Special thanks to my grandmother Nobu Adachi (*in memoriam*), who since my childhood taught me about the importance of studying. You will forever be my example of courage and perseverance.

My sincere gratitude to my supervisor, Alessandro Garcia, for all the confidence in me and for the enormous contribution to my professional growth. Your professionalism, dedication and enthusiasm are exemplary.

I thank all my coursemates at PUC-Rio, all my friends from LES and all my friends from my hometown. Without you company this journey would have been far more difficult.

I thank all the professors of the Department of Informatics at PUC-Rio for contributing to my education. I also thank all the employees of the department for their services.

I am also grateful to CNPq, FAPERJ and PUC-Rio for the financial support that made my PhD work possible.

To all of you, my sincere thanks.

Abstract

Barbosa, Eiji Adachi Medeiros; Garcia, Alessandro Fabricio (advisor). **Global-Aware Recommendations for Repairing Exception Handling Violations**. Rio de Janeiro, 2015. 213p. DSc Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Exception handling is the most common way of dealing with exceptions in robust software development. Exception handling refers to the process of signaling exceptions upon the detection of runtime errors and taking actions to respond to their occurrence. Despite being aimed at improving software robustness, software systems are still implemented without relying on explicit exception handling policies. Each policy defines the design decisions governing how exception handling should be implemented in a system. These policies are often not documented and are only implicitly defined in the system design. Thus, developers tend to introduce in the source code violations of implicit policies and these violations commonly cause failures in software systems. In this context, the goal of this thesis is to support developers in detecting and repairing exception handling violations. To achieve this goal, two complementary solutions were proposed. The first solution is based on a domain-specific language supporting the detection of violations by explicitly defining exception handling policies to be enforced in the source code. The proposed language was evaluated with a user-centric study and a case study. With the observations and experiences gathered in the user-centric study, we identified some language characteristics that hindered its use and that motivated new language constructs. In addition, the results of the case study showed that violations and faults in exception handling share common causes. Therefore, violations can be used to detect potential causes of exception-related failures. To complement the detection of exception handling violations, this work also proposed a solution for supporting the repair of exception handling violations. Repairing these violations requires reasoning about the global impact that exception handling changes might have in different parts of the system. Thus, this work proposed a recommender heuristic strategy that takes into account the global context of where violations occur to produce recommendations. Each recommendation produced consists of a sequence of modifications that serves as a detailed blueprint of how an exception handling violation can be removed from the source code. The proposed recommender strategy also takes advantage of explicit policy specifications, although their availability is not mandatory. The results of our empirical assessments revealed that the proposed recommender strategy produced recommendations able to repair violations in approximately 70% of the cases. When policy specifications are available, it produced recommendations able to repair violations in 97% of the cases.

Keywords

Exception handling; Recommender systems for software engineering; Exception handling policies; Repair of violations in exception handling.

Resumo

Barbosa, Eiji Adachi Medeiros; Garcia, Alessandro Fabricio. **Recomendações Globais para Reparação de Violações de Tratamento de Exceções**. Rio de Janeiro, 2015. 213p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Tratamento de exceções é o modo mais comum de lidar com erros no desenvolvimento de software robusto. Tratamento de exceções refere-se ao processo de sinalizar exceções quando erros em tempo de execução são detectados e de tomar ações para responder à ocorrência destas exceções. Apesar de objetivarem a melhoria da robustez de software, sistemas de software ainda são implementados sem se basear em uma política explícita para tratamento de exceções. Cada política define as decisões de projeto que governam como tratamento de exceções deve ser implementado num sistema. Tais políticas não são comumente documentadas e são apenas implicitamente definidas no projeto do sistema. Desta forma, desenvolvedores tendem a introduzir no código fonte violações das políticas implícitas e tais violações comumente causam falhas em sistemas de software. Neste contexto, o objetivo desta tese é apoiar desenvolvedores na detecção e reparação de violações de tratamento de exceções. Para atingir este objetivo, duas soluções complementares foram propostas. A primeira solução é baseada numa linguagem específica de domínio que apoia a detecção de violações ao definir explicitamente políticas de tratamento de exceções que devem ser obedecidas no código fonte. A linguagem proposta foi avaliada num estudo centrado no usuário e num estudo de caso. Com as observações e as experiências coletadas no estudo centrado no usuário, nós identificamos algumas características da linguagem que dificultavam o seu uso e que motivaram novos construtos. Além disso, os resultados do estudo de caso mostraram que violações e falhas costumam ter causas comuns. Portanto, violações de tratamento de exceção podem ser usadas para detectar causas de faltas relacionadas à exceções. Para complementar a detecção de violações, este trabalho também propôs uma solução para apoiar o reparo de violações de tratamento de exceções. Reparar estas violações requer raciocinar sobre o impacto global que mudanças em tratamento de exceções pode ter em diferentes partes do sistema. Desta forma, este trabalho propôs uma estratégia heurística de recomendação que leva em conta o contexto global onde violações ocorrem a fim de produzir recomendações. Cada recomendação produzida consiste em uma sequência de modificações que servem como um plano detalhado de como uma violação de tratamento de exceções pode ser removida do código fonte. A estratégia de recomendação proposta também se beneficia de especificações explícitas de políticas, embora sua disponibilidade não seja obrigatória. Os resultados das nossas avaliações empíricas revelaram que a estratégia de recomendação proposta produziu recomendações capazes de reparar violações em aproximadamente 70% dos casos. Quando especificações de políticas estão disponíveis, a estratégia produziu recomendações capazes de reparar violações em 97% dos casos.

Palavras-Chave

Tratamento de exceções; Sistemas de recomendação para a engenharia de software; Políticas de tratamento de exceções; Reparação de violações em tratamento de exceções.

Contents

List of Figures

List of Tables

1
Introduction

As software systems grow ubiquitous in our society, so does the need for robustness. Software robustness refers to the ability of a software system to remain in operation and deliver its services despite the occurrence of errors (IEEE, 1990). When the services delivered by software systems deviate from their specifications, it is said that *failures* occur (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). The state of a software system that enables the occurrence of a failure is called an *error* and the causes of errors are called faults (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). A *fault* is an incorrect or missing instruction or data definition in programs (IEEE, 1990). Accessing an invalid array index or using uninitialized variables are examples of faults in programs.

When software systems detect errors at runtime, they respond to service requests with exceptions (LEE and ANDERSON, 1990). An *exception* is an event signaled to indicate the impossibility of providing requested services (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). For this reason, the state of a software system is often inconsistent when exceptions are raised (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). If the system continues its execution in an inconsistent state, it may lead to additional exceptions or to failures. Therefore, it is important to avoid or mitigate the potential negative impact of exceptions in the system operation.

Exception handling is the most common way of dealing with errors during the development of software systems (BUHR, 2000, GARCIA et al., 2001, JAKOBUS et al., 2015). Exception handling refers to the process of detecting runtime errors, signaling exceptions upon the detection of these errors and taking actions to respond to the occurrence of these exceptions (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). This way, software robustness is improved if exception handling is designed and implemented properly.

Nowadays, developers design and implement exception handling using built-in exception handling mechanisms in programming languages (GARCIA et al., 2001, JAKOBUS et al., 2015). An *exception handling mechanism* is a set of language constructs to signal exceptions and to structure sets of actions responsible for coping with these exceptions (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). The importance of exception handling mechanisms in the development of software systems can be attested by their wide adoption in mainstream programming languages. Among the top-ten most widely adopted programming languages, nine languages provide built-in exception handling mechanisms (JAKOBUS et al., 2015).

However, despite being intended to improve software robustness, the causes of recurring failures in software systems are located in the exception handling code (CACHO et al., 2014a, CACHO et al., 2014b, COELHO et al., 2008, MARINESCU, 2011, MARINESCU, 2013, SAWADPONG et al., 2012). In particular, common failures are caused by global exceptions being left uncaught or being caught in the wrong place in the system (CACHO et al., 2014a, CACHO et al., 2014b, COELHO et al., 2008). *Global exceptions* are those raised and handled in different methods of a program (CACHO et al., 2008, CACHO et al., 2009, ROBILLARD and MURPHY, 1999, ROBILLARD and MURPHY, 2003). In fact, all failures reported in previous studies were caused by global exceptions (CACHO et al., 2014a, CACHO et al., 2014b, COELHO et al., 2008).

In this context, developers need to detect and repair the causes of exception-related failures. Otherwise, the exception handling code will compromise software robustness, instead of improving it.

1.1 Problem Statement and Limitations of Related Work

Exceptions are inherently global in software systems. As previously discussed, exceptions are raised when runtime errors are detected in software modules. Continuing program execution in a module where an error was detected may lead to failures in the system. For this reason, it is advisable to transfer program execution from the module where the exception was raised to another module, so that the error is confined to the module where it was detected. This is why most exceptions are raised and handled in different modules of a system. Therefore, dealing with global exceptions is central to the design and implementation of exception handling in any non-trivial software system.

In order to define and illustrate the problems tackled in this thesis, this section introduces an example extracted from Health Watcher (KULESZA et al. 2006, SOARES et al. 2002), an n-tier web-based system for controlling complaints about institutions inspected by health departments of city halls. This example was chosen because it is easy to understand and illustrates recurring failures whose causes are located in the exception handling code. This example was also chosen because it illustrates commonly reported failures caused by global exceptions (CACHO et al., 2014a, CACHO et al., 2014b, COELHO et al., 2008).

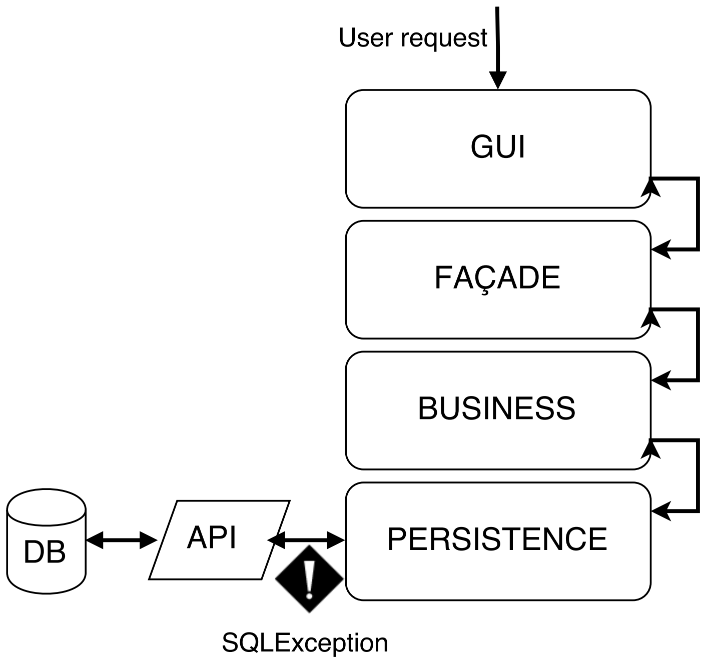


Figure 1.1: Health Watcher Structure

As one can observe in the diagram depicted in Figure 1.1, the structure of the Health Watcher system is separated into multiple tiers: *GUI*, *Façade*, *Business* and *Persistence*. The *GUI* tier is responsible for processing user requests. It processes user requests and delegates them to the *Business* tier through *Façade*. Then, the *Business* controls the complaints registered in the system by accessing the *Persistence* tier. The *Persistence* tier creates, reads, updates and deletes the complaints in the data base (DB) through an API. When the *Persistence* tier requests services to the API, the data base may not be able to provide the requested service correctly. For this reason, the API used to access the data base raises a `SQLException` to indicate this impossibility. In this case, the developer is expected to properly implement the exception handling behavior in charge of handling the `SQLException` raised by the data base API. Next, Listing 1.1 presents a simplified code snippet of the functionality responsible for saving a complaint in the system. The code

Listing 1.1: Exception-Related Failure in Health Watcher

```
1  // GUI
2  ComplaintGUI.insertSpecialComplaint(complaint){
3    try{ Facade.insertComplaint(complaint); }
4    catch( PersistenceException e ){ errorPage(); }
5  }
6  // FACADE
7  Facade.insertComplaint(complaint){
8    Complaint.insert(complaint);
9  }
10 // BUSINESS
11 Complaint.insert(complaint){
12   try{ Persistence.insertComplaint(complaint); }
13   catch( SQLException e)
14   { throw new RuntimeException(e); }
15 }
16 // PERSISTENCE
17 Persistence.insertComplaint(complaint){
18    // Access database.
19    // Db-API raises SQLException!
20 }
```

snippet depicts the source code that was deployed and caused a field failure, i.e., a failure observed in a production environment.

In the code snippet depicted in Listing 1.1, the developer allowed the `SQLException` to flow from within the *Persistence* tier (lines 17-20) to the *Business* tier (lines 10-15). In the *Business* tier, the developer decided to capture the `SQLException` with a *catch* block. A *catch* block delimits a set of statements implementing actions responsible for dealing with specific types of exceptions. Also, it declares an argument, which is usually an exception type. This argument is used as a filter to decide which exceptions the *catch* block captures. This filter usually follows sub-type compatibility rules of the programming language. Thus, a *catch* block captures any exception that is a subtype of its argument.

Within this *catch* block, the developer decided to invoke a *throw* statement. A *throw* statement explicitly raises an exception. When a *throw* statement is invoked from within a *catch* block, it is said that the captured exception is re-mapped to another type. In the previous example, the developer decided to re-map from `SQLException` to `RuntimeException` (line 14). However, the developer did not handle the re-mapped exception anywhere in the system. Thus, the re-mapped exception flowed through the methods in *Business*, *Façade* and *GUI* without being captured and left the boundaries of the Health Watcher system. Then, the web server that runs Health Watcher terminated its execution showing an error page displaying the uncaught exception. This way, an uncaught exception caused a failure by abruptly terminating the system execution.

Although one may think that the failure depicted in the previous example is not frequent because its cause seems simple, previous studies have shown that similar failures are commonly observed in software systems (CACHO et al., 2014a, CACHO et al., 2014b). In previous studies conducted by Cacho et al., re-mapped exceptions left uncaught was the most frequent cause of failures in the analyzed systems. In a sample of sixteen Java programs, 48% of the failures occurred because exceptions were re-mapped, but the re-mapped exceptions were not handled (CACHO et al., 2014a). Similarly, 38% of failures in sixteen C# programs were also caused by re-mapped exceptions being left uncaught (CACHO et al., 2014b). Uncaught exceptions usually cause severe failures because they make the whole system crash, making it unavailable to its users. Therefore, the failure depicted in the previous example is a frequent and severe exception handling-related problem.

1.1.1 Lack of Explicit Exception Handling Policies

Although exception handling is central to robust software development, most software systems still implement exception handling without relying on an explicit exception handling design (DELEMOS and ROMANOVSKY, 2001, KIENZLE, 2008). In this thesis, we refer to the set of design decisions governing how exception handling should be implemented in a system as the *exception handling policy* of the system. For the sake of brevity, from hereafter the terms “policy” and “exception handling policy” will be used indistinguishably in this thesis.

Developers participating in recent surveys reported that there exist exception handling policies in their systems, although not much effort is spent in documenting them (EBERT and CASTOR, 2013, EBERT et al., 2015). At best, these policies are partially documented in API documentation or source code comments. Most of the times these policies are not documented and exist as implicit rules in the source code (BUSE and WEIMER, 2008, THUMMALAPENTA and XIE, 2009). In fact, the lack of explicit exception handling policies is considered one of the main reasons why developers struggle to implement exception handling (SHAH et al., 2010).

Without explicit exception handling policies, most developers are not aware of how they should implement exception handling in their programs. Consequently, they implement exception handling in an *ad-hoc* manner, probably introducing in the source code violations to implicit policies. An *exception handling violation* occurs when the implemented exception handling code does not comply with the exception handling policy of a system. From hereafter, the terms "exception handling violation" and "violation" will be used indistinguishably throughout this thesis.

These violations may have negative consequences in software systems. In the code snippet depicted in Listing 1.1, for example, there exists a *catch* block declaring the `PersistenceException` type in the *GUI* tier (line 4). The developer was expected to re-map from `SQLException` to `PersistenceException` and then handle the re-mapped exception with this existing *catch* block in *GUI*. However, the developer violated this implicit policy by implementing an incorrect re-mapping from `SQLException` to `RuntimeException`. This violating re-mapping originated an uncaught exception, which caused a field failure. In fact, recent studies have been reporting exception handling violations as frequent causes of exception-related failures: all failures in (COELHO et al., 2008), 90% of failures in (CACHO et al., 2014b), 85% of failures in (CACHO et al., 2014a) and 47% of failures in (EBERT et al., 2015) were caused by violations.

Also, the lack of explicit policies hinders the detection of exception handling violations. Without knowing how exception handling was expected to be implemented, developers cannot identify which parts of the code are not implemented as intended. Since exception handling code is rarely exercised during program execution and exceptional situations are poorly tested (FU et al., 2005,SINHA and HARROLD, 2000), exception handling violations remain dormant in the source code. For this reason, these violations are only discovered later when they cause field failures (FU et al., 2005,SINHA and HARROLD, 2000).

In this context, the first problem tackled by this thesis is stated as follows:

The lack of explicit exception handling policies in software systems.

Even if software designers and developers are keen on explicitly defining exception handling policies for their systems, there is still no proper support for that. Currently, mainstream programming languages provide built-in exception handling mechanisms. However, these mechanisms were conceived to implement error handling behavior in programs. They are not intended to define policies expressing how exception handling should be implemented. At best, developers can express part of their exception handling design decisions in the names of exception types. For example, the `SQLException` type, as its name suggests, should be used to represent database-related errors. Implicitly, this is an exception type that should be used by modules implementing database-related features. Thus, based on the name of the exception type, developers can infer which modules raised exceptions of this type. However, one cannot assure that exceptions of this type will only be raised by persistence-related modules. Similarly, one cannot assure that persistence-related modules will only raise exceptions of this type. Also, one cannot express where exceptions of this type are intended to be re-mapped and handled, for example. And this type of design decision is important to exception handling, as shown in the previous example depicted in Listing 1.1. Developers may even try to express these decisions as source code comments, but comments do not enforce policy adherence nor aid the detection of violations in the source code.

Currently, there exist solutions aimed at specifying and verifying exception handling design rules (ABRANTES and COELHO, 2015, CACHO et al., 2008, SALES and COELHO, 2011, SILVA and CASTOR, 2013). All these solutions rely on the assumption that for designing exception handling it is enough to specify the places where exception should be raised and handled in the source code. However, these are only part of the design decisions involved in dealing with exceptions. Exceptions are commonly captured and re-mapped along their propagation paths (FU and RYDER, 2007). And recent studies showed that recurring uncaught exceptions occurred due to the introduction of *catch* blocks that re-map exceptions (CACHO et al., 2014a, CACHO et al., 2014b). No current solution takes into account these properties of global exceptions, that is, they do not support the specification of the places where global exceptions are supposed to be re-mapped. Therefore, current solutions still provide limited support for the specification and verification of exception handling policies.

1.1.2 Difficulty in Repairing Violations

Exception handling violations must be repaired to avoid the risk of causing failures. The repair of exception handling violations refers to performing modifications in the source code to make it policy-compliant. However, repairing exception handling violations is a difficult and error-prone task. The difficulty in repairing exception handling violations stems from the complexity in performing modifications in the error handling behavior of a system. These modifications often deal with global exceptions, which require non-local reasoning about the possible impact of these exceptions. Global exceptions may impact any method in the call-chain of methods through which they traverse. While performing changes in a given method m, there exists the possibility of exceptions coming from any method directly or transitively invoked by m. Similarly, there is also the possibility of exceptions flowing to any method that directly or transitively invokes m. Moreover, any non-trivial software system has complex call-chains of methods and global exceptions that traverse through multiple methods. Thus, modifying the source code that deals with global exceptions requires reasoning about their impact in many different places of the system.

Consider for example the violation that occurs in the `Complaint.insert` method depicted in Listing 1.1 (lines 11-15). It should be noted that the code snippet depicted in Listing 1.1 is only a simplified version of the real source code. Considering the call-chain of methods actually implemented in the system, the `Complaint.insert` directly invokes 15 methods. If transitive method invocations are considered, this number exceeds the hundreds. Also, there exist 56 different methods transitively calling `Complaint.insert`. Therefore, developers have to reason about a large number of possible places where exceptions can come from and where they can flow to. Failing to properly reason about the possible impacts of global exceptions in these places may introduce faults in the source code (CACHO et al., 2008, CACHO et al., 2009, COELHO et al., 2008). Developers can inadvertently catch exceptions in the wrong place, implement incorrect re-mappings, miss the implementation of expected re-mappings or miss the implementation of expected handlers (COELHO et al., 2008, CACHO et al., 2014a, CACHO et al., 2014b, EBERT et al., 2015).

The difficulty in repairing violations is worsened when explicit policies are not available because developers do not know which violations may be causing the failure. They also do not know how they should modify the source code to make it policy-compliant. Thus, they follow an *ad-hoc* strategy to identify and repair the causes of the failures. Following *ad-hoc* strategies, developers often introduce other violations in the source code while trying to repair existing ones (CACHO et al., 2014a, CACHO et al., 2014b). In fact, developers introduce exception handling violations in the source code even when doing simple modifications, such as adding method invocations to existing *catch* blocks (CACHO et al., 2014a, CACHO et al., 2014b).

Consider for now that developers tried to repair the cause of the failure depicted in Listing 1.1 without being aware of the implicit policy of the system. This failure was caused by an uncaught exception originated from the re-map performed in a method located in the *Business* tier. Once developers trace back the uncaught exception to the place where it was originated, they may infer that the cause of the failure relates to the re-mapping. Then, developers may try to repair it in different ways. One developer may try to repair it by adding a *catch* block to handle the uncaught `RuntimeException` in one of the methods in the tiers above the *Business* tier. However, this *catch* block would also handle other exceptions that are subtypes of `RuntimeException`, which includes `NullPointerException`, `ArithmeticException`, and the like. This way, this *catch* block would handle exceptions related to other faults, rather than only the one detected in the *Persistence* tier. And handling these exceptions in the incorrect place would probably lead to other failures.

Alternatively, another developer may try to repair it by removing the *throw* statement that originated the uncaught exception. That is, instead of re-mapping from `SQLException` to `RuntimeException`, this developer would handle the `SQLException` in the *Business* tier. However, this decision would hide the fault that originated the `SQLException` in the first place. Hiding the original fault would allow the system to continue its execution without correctly executing the persistence service, which would actually characterize another failure. In fact, when developers follow *ad-hoc* strategies to repair violations related to global exceptions they run the risk of introducing other violations, possibly leading to new failures (CACHO et al., 2014a, CACHO et al., 2014b). As tests are often not properly defined for exception handling (FU et al., 2005, SINHA and HARROLD, 2000), it is unlikely that the developer would review possible failures before the system is deployed to a production environment. If this violation leads to a field failure, the system would persist inconsistent data, possibly causing other failures when this data is retrieved. Persisting inconsistent data could even cause failures in other systems using the same database. Therefore, this violation would have subtle and severe negative consequences to the system.

Even if developers were aware of the implicit policy of the system, repairing the cause of the failure depicted in Listing 1.1 would still be a difficult task. First, developers would need to identify the exception handling violations in the source code. Then, they would need to figure out how they should modify the source code to make it policy-compliant. In the previous example, developers would have to identify that the *Persistence* tier should not have propagated `SQLException` to the upper tier. Instead, the *Persistence* tier should have re-mapped from `SQLException` to `PersistenceException`. Then, developers would have to realize that the re-mapped `PersistenceException` was supposed to be handled in the *GUI* tier. Developers would also have to identify that the *Business* tier was not supposed to re-map from `SQLException` to `RuntimeException`.

After identifying the exception handling violations in the source code and realizing how the exception handling is supposed to be implemented, developers would have to modify the source code to make it policy-compliant. To make the source code policy-compliant, developers would have to remove the violation in the *Persistence* tier related to the absence of a re-mapping and also the violation in *Business* tier related to an existing but incorrect re-mapping. To do so, first, developers would have to modify the method in the *Persistence* method tier to remap from `SQLException` to `PersistenceException`. Then, they would have to propagate the `PersistenceException` to the *GUI* tier, where a proper method would handle it. Propagating the exception from *Persistence* to *GUI* requires inspecting all the methods through which the exception traverse to assure that no method before *GUI* is catching it along the propagation path. This requires inspecting 56 methods distributed in 9 classes pertaining to 4 modules in the system.

In this context, the second problem tackled by this thesis is stated as follows:

The difficulty in repairing exception handling violations in the source code.

Currently, there is still no support for the repair of exception handling violations. Many solutions supporting exception handling are aimed at visualizing how exceptions propagate through methods in the source code (CHANG et al., 2001,FU and RYDER, 2007,ROBILLARD and MURPHY, 1999,ROBILLARD and MURPHY, 2003,SHAH, GÖRG and HARROLD, 2008a). However, visualization-based solutions do not provide visual cues of where violations are located in the source code, nor assists in the comprehension of which modifications must be performed in the source code to repair the violations. At best, these solutions may assist the comprehension of the source code before developers try to repair violations.

Solutions aimed at specifying and verifying exception handling design rules can only detect the places where specifications are violated (ABRANTES and COELHO, 2015,CACHO et al., 2008,SALES and COELHO, 2011,SILVA and CASTOR, 2013). They do not assist developers in deciding which modifications in the source code must be performed to actually repair these violations. Without this type of support, developers have to repair these violations on their own, often performing modifications that introduce other violations while they try to repair existing ones, as previously discussed.

There are also solutions supporting exception handling based on recommender systems (BARBOSA et al., 2012, BARBOSA et al., 2012a, RAHMAN and ROY, 2014), but they are not aimed at supporting the repair of exception handling violations. These solutions are aimed at assisting the implementation of exception handlers. Therefore, they are not aware of the existence of exception handling violations in the source code, let alone support their repair. Finally, in the software architecture literature, there is one solution aimed at assisting developers in repairing architectural violations, some of which are related to exception handling (TERRA et al., 2015). The main limitation of this solution when employed to the repair of exception handling violations is its unawareness of the global impact that exceptions might have. Unaware of the global effect of exceptions, this solution may be able to repair a given exception handling violation, but the changes performed may actually introduce other violations in the source code. This is similar to what we previously discussed when we presented the two strategies developers could follow without policies.

1.2 Goal and Research Questions

As discussed in the previous section, the lack of explicit exception handling policies brings negative consequences to software systems. It contributes to the introduction of exception handling violations in the source code and also hinders the detection of these violations. In addition, when exception handling violations are detected in the source code, they must be repaired. However, repairing exception handling violations requires extensive reasoning about the impact of global exceptions, which is a far from trivial task. If developers fail to detect and properly repair exception handling violations, they run the risk of letting such violations cause failures.

In this context, the goal of this thesis is stated as follows:

> ***Goal.*** *Support the detection and repair of exception handling violations in the source code of software systems.*

To achieve this goal, two complementary solutions were proposed. The first proposed solution supports the detection of exception handling violations in the source code. Supporting the detection of violations in the source code requires tackling the problem of the lack of explicit exception handling policies.

Thus, by explicitly defining exception handling policies and checking their conformance in the source code, the detection of violations is addressed. In this context, our first research question is stated as follows:

> ***RQ1.*** *How to support the definition and checking of exception handling policies in the source code?*

To address the research question RQ1, we have designed a domain-specific language (VANDEURSEN et al., 2000, FOWLER, 2010) to specify and verify exception handling policies. Developers can benefit from the proposed language by using it to explicitly define policies governing how exception handling should be implemented in their systems. They can also use it to detect and localize where exception handling violations occur in the source code. Therefore, the proposed language fulfills the first part of the goal of this thesis. Chapter 4 presents the language proposal and the studies conducted to evaluate it.

To complement this first solution, this work also proposed a solution for supporting the repair of exception handling violations. This led to our second research question:

> ***RQ2.*** *How to support the repair of exception handling violations in the source code?*

The second research question (RQ2) aimed at investigating a solution to support the repair of exception handling violations. Addressing the research question RQ2 required tackling the difficulty of repairing violations related to global exceptions. To do so, we proposed a recommender heuristic strategy that takes into account the global context of where exception handling violations occur. This global context encompasses the whole call-chain of methods where the violation is located. This way, the proposed heuristic strategy considers the possible impacts of global exceptions in order to provide recommendations on how to repair violations in the source code. Therefore, the proposed recommender heuristic fulfills the second and final part of the goal of this thesis. The heuristic proposal and the study conducted to evaluate it are presented in Chapter 5.

Documenting explicit exception handling policies with the proposed language supports the detection of violations in the source code. After detecting these violations, the proposed recommender heuristic supports their repair. In addition, the recommender heuristic leverages on explicit policy specifications to improve its effectiveness. Therefore, developers will be better-off if they use the two proposed solutions combined.

Finally, it is worth discussing why this thesis aimed at supporting the detection and repair of exception handling violations, instead of directly detecting and repairing exception handling faults. In fact, our initial goal was to support the detection and repair of exception handling faults. Before investigating solutions for this type of support, we investigated what categories of exception handling faults occur. This study is presented in Chapter 3. By better understanding what categories of exception handling faults occur, we thought that it would be possible to support their detection and repair.

However, during this study on exception handling faults, we observed that most of these faults were not confined to specific structural patterns in the source code, such as empty or generic *catch* blocks. Actually, we observed an ambiguity in some structural patterns in exception handling code: whereas some faults occurred due to a specific structural pattern, other faults were repaired using the same structural pattern. For example, there were faults that occurred due to generic *catch* blocks, whereas other faults were repaired using generic *catch* blocks. In the example depicted in Listing 1.1, the failure was caused by a re-mapping, but it was also repaired by implementing another re-mapping. Therefore, it would be difficult to support the detection of exception handling faults by only analyzing structural patterns in the source code, as explored in programs that implement exception handling using idioms based on returning error values (BRUTNIK et al., 2006). This type of solution is more appropriate to programs implemented in programming languages without a built-in exception handling mechanism. By only reviewing the source code of the system, it is impractical to decide whether a given element in the exception handling code is faulty or not. One would need a proper specification defining the intended exception handling design. In other words, it became clear to us the need for explicit policies in order to properly analyze the exception handling implementation. For this reason, we focused on supporting the definition of explicit exception handling policies, as well as on supporting the detection and repair of exception handling violations in the source code.

1.3 Thesis Outline

This introductory chapter portrayed an overview of this thesis. The remainder of the thesis is structured as follows. Chapter 2 gives an overview of the basic concepts explored in this thesis and also summarizes the main works related to ours. Chapter 3 presents the preliminary study conducted to investigate what types of exception handling faults occur in software systems. Chapter 4 presents a domain-specific language for specifying and verifying exception handling policies in programs. Chapter 5 presents a recommender heuristic strategy that assists the repair of exception handling violations. Finally, Chapter 6 concludes this thesis by summarizing the research contributions achieved, making final considerations and pointing to directions for future research.

2 Background

This chapter outlines the terminology adopted throughout this thesis (Section 2.1). It also presents a series of previous empirical studies conducted to investigate how exception handling is being implemented in software systems (Section 2.2). Finally, it presents current solutions proposed for supporting exception handling (Section 2.3).

2.1 Basic Terminology

A *software system* consists of a set of modules that cooperate with each other to deliver services to an external environment (LEE and ANDERSON, 1990). A *software module* is a software element that encapsulates a set of functionalities and provides services through an interface (LEE and ANDERSON, 1990). When a software system delivers its services according to their specifications, it is said that the system delivers its services correctly (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). On the other hand, when the services delivered by a software system deviate from their specifications, it is said that a *failure* occurs (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). Failures occur either because the system implementation does not comply with its specification or because the specification does not describe the system's services adequately. The state in which the services of a system deviate from their specifications is called an *error* (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). In other words, errors are the states that enable the occurrence of failures. And the cause of an error is called a fault (AVIZIENIS et al., 2004, LEE and ANDERSON, 1990). *Faults* are incorrect or missing instructions or data definitions in programs (IEEE, 1990).

Robust software systems must remain in operation despite the occurrence of errors (IEEE, 1990). To do so, they must be able to detect errors during runtime and take actions to confine the consequences of errors, allowing the system to continue in operation. When a software system is robust and is also able to deliver its services correctly after detecting errors, this system is also fault tolerant (LEE and ANDERSON, 1990). Fault tolerance is achieved by detecting errors and responding to these errors by taking actions for recovering the system from inconsistent states (LEE and ANDERSON, 1990).

Recovering from inconsistent states can be achieved with two broad categories of techniques: backward error recovery and forward error recovery (LEE and ANDERSON, 1990). In backward error recovery, the state of a system is restored to a recovery point upon the detection of errors. Recovery points are previous and non-erroneous states of the system (LEE and ANDERSON, 1990). They are recovered by undoing the effects of operations performed since the last recovery point was established, or by saving and reloading previous states. In forward error recovery, the state of a system is restored without reversing previous operations, nor saving previous states (LEE and ANDERSON, 1990). Forward error recovery is performed by finding a safe state for the system. After an error is detected, safe states can be achieved by reconfiguring the system, using redundancy to fix corrupted data or replacing an erroneous value with a default value that will have a benign effect, for example.

Backward error recovery techniques can be expensive and impose performance penalties to programs because they require saving previous states of the system or a log of the operations performed (LEE and ANDERSON, 1990). In addition, there is no guarantee that after recovering the system to a recovery point the error will not occur again. If the error was caused by a fault in the source code of the system, the error will likely occur again if the system is recovered and then executes the same portion of the code where the fault is located. Forward error recovery techniques, on the other hand, impose fewer overheads to programs (LEE and ANDERSON, 1990). In addition, if the error was caused by a fault in the source code, forward error recovery techniques are able to avoid executing the portion of the code where the fault is located. For these reasons, forward error recovery is more employed in software systems than backward error recovery. Exception handling, which is the focus of this thesis, is one of the main instruments for implementing forward error recovery in software systems.

2.1.1 Exception Handling

Software modules receive service requests from the external environment and produce responses. The diagram depicted in Figure 2.1 presents the structure of robust software modules adapted from (LEE and ANDERSON, 1990). This abstract structure serves as a conceptual framework for the design and implementation of robust software modules. As one can observe in the

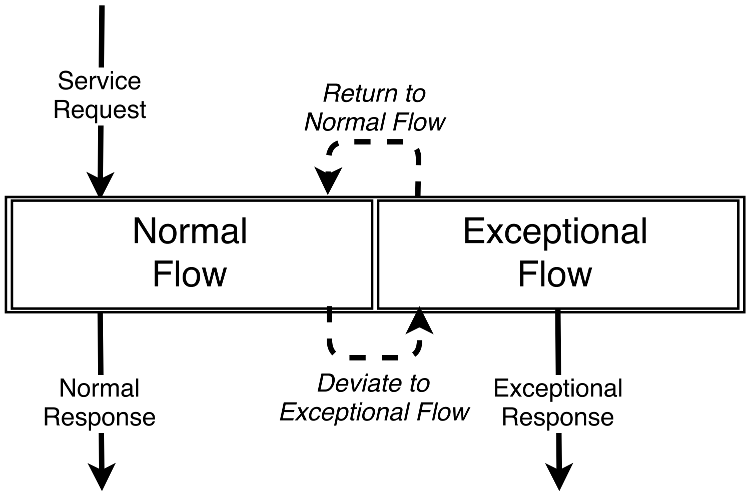


Figure 2.1: Internal Structure of a Robust Software Module

diagram depicted in Figure 2.1, the execution flow of a robust module can be internally separated into normal and exceptional flows (LEE and ANDERSON, 1990). In addition, the responses from a robust module can be separated into two distinct categories: normal responses and exceptional responses (LEE and ANDERSON, 1990). When robust modules receive service requests and they are able to provide their services correctly, then they execute their normal flow and produce normal responses. *Normal responses* correspond to the results of correct services and the *normal flow* comprises the actions that a robust module implements to provide its services correctly (LEE and ANDERSON, 1990).

In addition, when robust modules detect errors during the execution of their normal flow, they must take actions to respond to these errors. To do so, robust modules deviate their execution flow from the normal flow to the exceptional flow. The *exceptional flow* comprises the actions that a robust module implements to respond to errors (LEE and ANDERSON, 1990). If robust modules are able to properly confine the consequences of an error, then their execution flow resumes from the exceptional flow to the normal flow. Otherwise, they respond to service requests by producing exceptional responses. *Exceptional responses* are produced to indicate that requested services cannot be correctly provided. An exceptional response is also referred to as an *exception* (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975).

Software modules typically signal exceptions when errors are detected during runtime. For this reason, the state of a software module is often inconsistent when exceptions are signaled (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). If the module continues its execution in an inconsistent state, it may lead to additional exceptions or even to failures. For this reason, it is important to avoid or mitigate the potential negative impact of exceptions in the system operation. Otherwise, software robustness is compromised.

Exception handling is the most common way of dealing with exceptions during the construction of robust software systems (BUHR, 2000, GARCIA et al., 2001, JAKOBUS et al., 2015). *Exception handling* refers to the process of identifying runtime errors, signaling exceptions upon the detection of these errors and taking actions to respond to the occurrence of these exceptions (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). These actions should, ideally, confine the consequences of errors and allow the system to remain in operation. This way, the proper design and implementation of exception handling improves software robustness.

In order to support developers in implementing robust software systems, most mainstream programming languages provide built-in exception handling mechanisms (GARCIA et al., 2001, JAKOBUS et al., 2015). An exception handling mechanism is a set of language constructs to (i) represent exceptions in software systems, (ii) signal the occurrence of these exceptions, (iii) structure sets of actions responsible for coping with exceptions and (iv) activate these actions upon the detection of exceptions (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975). The set of actions responsible for coping with exceptions is also referred as an *exception handler*. Programming languages with built-in exception handling mechanisms also provide runtime support for: (i) deviating the execution flow of a program from its normal to its exceptional flow when exceptions are raised and (ii) resuming from the exceptional to the normal flow when exceptions are handled (BUHR, 2000, CRISTIAN, 1989, GOODENOUGH, 1975).

In the context of programming languages, there are different variants of how exception handling mechanisms can be implemented (BUHR, 2000, GARCIA et al., 2001). We observed in a recent study that among the 54 most widely adopted programming languages with built-in exception handling mechanisms, 53 follow the *try-catch-throw* model (JAKOBUS et al., 2015). This model relies on the definition of guarded scopes with *try* blocks, exception handlers with *catch* blocks and exceptions raisers with *throw* statements. The *try* blocks delimit a portion of the code that may raise exceptions, but is guarded from their occurrence by *catch* blocks. The *catch* blocks delimit a portion of the code responsible for dealing with specific types of exceptions. When a *throw* statement is invoked to raise an exception, the program execution is transferred from the place where *throw* statement was invoked to the first enclosing *catch* block able to capture this exception. After the *catch* block is executed, the program continues its normal flow.

This thesis focuses on exception handling mechanisms that follow the *try-catch-throw* model. In particular, it focuses on the mechanism implemented by the Java programming language for two main reasons. First, Java is the most adopted programming language with a built-in exception handling mechanism following the *try-catch-throw* model (JAKOBUS et al., 2015). Second, the characteristics of the Java exception handling mechanism subsume the characteristics of the mechanisms in most mainstream programming languages (GARCIA et al., 2001). Thus, the Java exception handling mechanism is representative of mechanisms most frequently used by developers nowadays to achieve software robustness. The next section describes the exception handling mechanism of the Java programming language.

2.1.2 Exception Handling in Java

This section describes the exception handling mechanism of the Java programming language, as described in *The Java Languages Specification: Java SE 6* (JSL-6). The next sections describe the main features of the exception handling mechanism of Java.

Exception Representation

Exceptions are represented as objects in Java. More specifically, all exceptions in Java are instances of the class `Throwable` or one of its subclasses (JSL-6). The `Throwable` class is the root of the exception hierarchy tree in Java. The diagram in Figure 2.2 depicts the partial exception hierarchy tree in Java. The `Throwable` class has two direct subclasses: `Error` and `Exception`. Exceptions that are instances of the `Error` class, or of one of its subclasses, represent serious problematic conditions external to the application (JSL-6). For this reason, these exceptions should not be handled at the application level. An example of a problematic condition external to the application occurs when the Java Virtual Machine runs out of resources necessary for it to continue operating (e.g. `OutOfMemoryError`).

The `RuntimeException` class is a direct subclass of the `Exception` class. The `RuntimeException` class and its subclasses represent exceptions that are internal to the application (JSL-6). These exceptions are typically raised when semantic constraints of the Java programming language are infringed. Examples of such infringements are attempts to access null point-

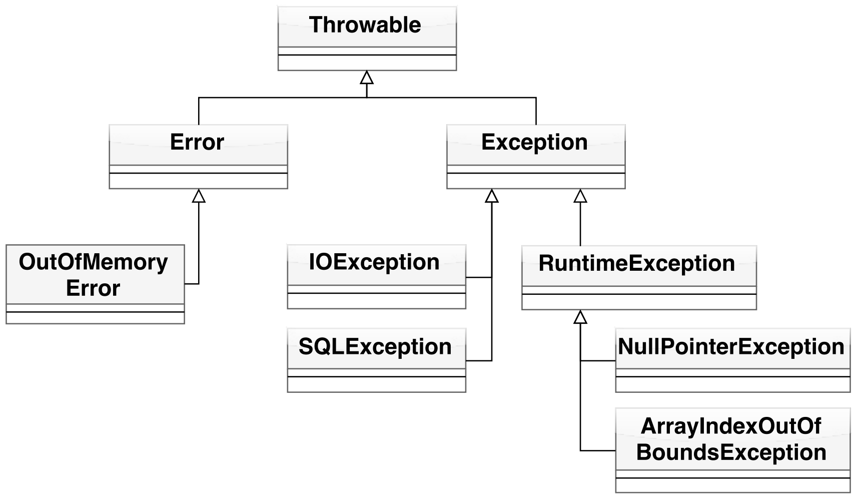


Figure 2.2: Exception Hierarchy Tree in Java

ers or index outside the bounds of an array. These infringements result in `NullPointerException` and `ArrayIndexOutOfBoundsException`, respectively. For this reason, client code cannot reasonably be expected to recover from or to handle these exceptions in any way. Therefore, measures should be taken in the source code to prevent the occurrence of exceptions that are instances of `RuntimeException` or one of its subclasses.

The `Exception` class and its subclasses, except those that inherit from the `RuntimeException` class, represent exceptions that are internal to the application and are typically related to recoverable conditions (JSL-6). In other words, they represent exceptions that the application is able to recover from. For example, an instance of the `SQLException` class may be raised to indicate a transient problem in a database server. If the application performs some recovery actions and retries the operation that previously raised the exception, then this operation may be able to succeed.

Checked vs. Unchecked Exceptions

In Java, exceptions can be classified as either checked or unchecked exceptions. The *unchecked* exceptions are those that are subtypes of the `RuntimeException` and `Error` classes (JSL-6). All the other exceptions are *checked* exceptions (JSL-6). That is, the checked exceptions comprise all exceptions that are subtypes of the `Throwable` class other than the subtypes of `RuntimeException` and `Error` classes.

Methods that raise checked exceptions are bound to the *"Catch or Specify Requirement"* (JSL-6). This requirement states that methods raising checked exceptions must either capture these exceptions with a *catch* block or specify these exceptions in their signature with the *throws* clause. The *throws* clause defines in the signature of a method its exceptional interface, i.e., the list of exceptions that flows through the boundaries of this method. Moreover, a method calling another method with checked exceptions in its exceptional interface is also bound to the *"Catch or Specify Requirement"*. The Java compiler statically verifies at compile-time if methods adhere to the *"Catch or Specify Requirement"*. If a method fails to adhere to this requirement, the Java compiler signals an error. This compile-time verification for checked exceptions is aimed at reducing the number of exceptions that are not properly handled by developers (JSL-6).

The static reliability check to verify the *"Catch or Specify Requirement"* is not performed for the unchecked exceptions. The declaration of unchecked exceptions in the exceptional interface of a method is not mandatory. According to the Java specification, exceptions that are subtypes of the `Error` class are not required to be declared in exceptional interfaces because they can occur at many points in the program and recovering from them is difficult or impossible (JSL-6). Requiring the declaration of these exceptions would make programs excessively cluttered. Similarly, the designers of the Java programming language judged that having to declare exceptions that are subtypes of the `RuntimeException` in exceptional interfaces would not aid in establishing the correctness of programs (JSL-6). Many of the operations in Java can result in `RuntimeExceptions`, so requiring such exceptions to be declared would only be an extra burden to developers.

Although not mandatory, unchecked exceptions may still be declared in exceptional interfaces. Declaring unchecked exceptions in the exceptional interfaces of methods does not impose any contract to callee methods. Bloch argues against the declaration of unchecked exceptions in exceptional interfaces (BLOCH, 2008). Bloch argues that developers must be aware of which exception types are checked and which are unchecked, as their responsibilities differ in these two cases. For this reason, the exceptional interface of a method provides should help developers distinguish checked exceptions from unchecked. Therefore, only checked exceptions should be declared in the exceptional interface of a method.

Guarded Scopes, Exception Handlers and Terminating Actions

A guarded scope delimits a portion of code guarded from the occurrence of exceptions. An exception handler is a set of actions responsible for coping with specific exceptions detected during runtime. Guarded scopes and exception handlers are defined with *try* blocks and *catch* blocks, respectively, as depicted in the following generic structure:

```
try{ S }
catch( E_1 ) { H_1 }
catch( E_2 ) { H_2 }
```

The *try* block guards a sequence of statements S from occurrences of exceptions by preventing that certain exceptions flow out of the block. To prevent that exceptions flow out of a *try* block, a sequence of *catch* blocks is attached to it. Each *catch* block may capture specific exceptions flowing out of the *try* block.

A *catch* block is defined in terms of an argument E_n and a sequence of statements H_n. The argument of a *catch* block is a type used as a filter of which exceptions may be captured. In Java, this filter follows sub-type compatibility rules of the programming language (JSL-6). Thus, a *catch* block declaring the type E_n as its argument captures any exception of this type and also exceptions that are subtypes of E_n. When a *catch* block captures an exception that is a subtype of its argument, it is said that the exception was caught by subsumption. Moreover, the sequence of statements H_n implements the set of actions that actually cope with an exception.

The terminating actions are delimited by *finally* blocks. These blocks may be attached to *try* blocks, as depicted in the following generic structure:

```
try{ S }
catch( E ) { H }
finally { F }
```

The *finally* block delimits a sequence of statements F that is always executed when the control leaves the *try* block. The control can leave a *try* block when all statements in the *try* block are executed normally. Control can also leave as a result of the execution of a *break*, *continue* or *return* statement, or of an exception flowing out of the *try* block. When an exception flows out of the *try* block and a *catch* block captures this exception, the *finally* block is executed after the execution of the *catch* block.

In other words, the *finally* block is always executed, regardless of whether the *try* block terminates normally or exceptionally. In practice, *finally* blocks can be used to avoid that a transfer of control accidentally bypasses a given sequence of statements. For this reason, *finally* blocks are often used to perform actions that must be executed every time a sequence of statements terminates its execution, even in the occurrence of exceptions. The following code snippet exemplifies the definition of guarded scopes, exception handlers and terminating actions:

```
public Data readData( String sql ){
   Connection conn = DbUtils.getConnection();
   Statement stmt = null;
   try{
      stmt = conn.createStatement();
      ResultSet set = stmt.executeQuery( sql );
      // process set
   }
   catch( SQLException e ) { conn.rollback(); }
   finally { if (stmt != null) stmt.close(); }
}
```

In the previous code snippet, there is a definition of a *try-catch-finally* block within the `readData` method. It is worth noticing that guarded scopes in Java are defined at the block-level. That is, *try* blocks are attached to blocks of statements in the source code. In the previous examples, the method invocations `conn.createStatement()` and `stmt.executeQuery( sql )` may result in exceptions of the `SQLException` type. These invocations are guarded by the *try* block, to which a *catch* block is attached. There is also a *finally* block attached to the *try* block. This *finally* block is responsible for invoking the `stmt.close()` method in order to release an allocated resource. If a `SQLException` occurs during the execution of `conn.createStatement()` or `stmt.executeQuery( sql )`, then the `conn.rollback()` method is invoked within the *catch* block to undo the changes performed in the database and release any lock held by the database connection. After executing the *catch* block, the *finally* block is executed to close opened resources. If no `SQLException` is raised, then all the statements within the *try* block are executed, followed by the execution of the statements within the *finally* block.

Exception Raisers

Exceptions are raised with the *throw* statement. The result of raising an exception is an immediate transfer of control from the normal flow to the exceptional flow of a program. The Java virtual machine halts the program execution and starts the search for a suitable handler. Suitable handlers are *catch* blocks able to capture the raised exception.

The search for a suitable handler starts from the place where the exception is raised. If the exception is raised from a statement outside a *try* block, then the call-stack is unwound until a *try* block is found. When a *try* block is found or when the exception is raised from within a *try* block, a suitable handler is searched in the list of *catch* blocks attached to this *try* block. The runtime type of the raised exception is compared with the exception types in each *catch* block. For the first matching *catch* block, its sequence of statements is executed. When all statements of a *catch* block are executed without raising any exception, it is said that the exception is handled. After an exception is handled, the execution flow returns to the normal flow in the first syntactic unit after the executed *catch* block.

If no matching *catch* block is found in the list of *catch* blocks attached to a *try* block, then the call-stack continues to be unwound until another *try* block is found. If the whole call stack is unwound and no matching *catch* block is found, then the thread that detected the exception is terminated (JSL-6).

Exception Propagation

The exception handling mechanism implemented by Java provides a hybrid design solution for exception propagation. In Java, unchecked exceptions are implicitly propagated. That is, unchecked exceptions are automatically propagated from the method where they are raised to the methods where proper handlers are found, possibly traversing through intermediate methods. Intermediate methods are those in an exception propagation path situated between the place where the exception is raised and the places where the exception is handled.

Checked exceptions, on the other hand, are explicitly propagated. That is, checked exceptions are only propagated when they are declared in the exceptional interface of methods. If a checked exception occurs in the context of a method and it is neither handled by a *catch* block nor declared in the *throws* clause of the method, then the Java compiler signals a compilation error.

Exception Re-throwers and Re-mappers

After being raised, exceptions usually traverse through different intermediate methods on the call stack before they are handled. In some cases, it may be necessary to implement partial handling actions in intermediate methods. In these cases, intermediate methods need to capture the exception, implement partial handling actions and re-throw the caught exception to its immediate callers.

In Java, there is no specific command to re-throw an exception. Instead, a re-throw is performed as follows:

```
try { S }
catch( E1 e ){
  //perform handling actions
  throw e;
}
```

As one can observe in the previous code snippet, a re-throw is implemented in Java with a *throw* statement within a *catch* block that explicitly receives as its argument the same exception captured by the *catch* block.

In addition to scenarios in which intermediate methods need to re-throw the caught exception, there are also scenarios in which they need to re-map the caught exception. An exception re-mapping occurs when an exception different from the exception caught by the *catch* block is raised from within the *catch* block. Java does not provide a specific construct to perform exception re-mappings. The following generic code depicts how exception re-mappings can be implemented in Java:

```
try { S }
catch( E1 e ){ throw new E2(e); }
catch( T1 e ){ throw new T2(); }
```

In the previous code snippet, the first *catch* block captures exceptions that are subtypes of E_1 and raises an exceptions of type E_2. In this case, it is said that the exception is re-mapped from E_1 to E_2. Notice that the caught exception is wrapped in the re-mapped exception. This practice can be used to store the original exception in the re-mapped exception. In fact, this is a common practice and most of Java built-in exception classes have constructors that take an exception as parameter. The second *catch* block in the previous code snippet re-maps from T_1 to T_2 without wrapping the caught exception in the re-mapped exception. Exception re-mappings can be implemented in both manners; developers are free to decide which manner suits them best.

2.2 Exception Handling in Practice

This section provides an overview of the empirical studies related to the theme of this thesis. We present empirical studies we conducted in collaboration with other researchers, as well as studies conducted by other research groups. Next, Section 2.2.1 presents empirical studies that assessed how exception handling is implemented in programs. Section 2.2.2 presents studies conducted to analyze the occurrence of faults located in the exception handling code.

Table 2.1: Types of Exception Handlers

	Handler Type	Libraries	Stand-alone	Server-apps	Servers
.NET	Alternative Configuration	28%	11%	23%	25%
	Empty	21%	21%	11%	25%
	Log	15%	33%	42%	14%
	Return	3%	15%	14%	0%
	Throw	6%	14%	8%	16%
Java	Alternative Configuration	3%	4%	3%	7%
	Empty	11%	15%	17%	13%
	Log	28%	57%	38%	41%
	Return	9%	15%	7%	11%
	Throw	44%	5%	30%	23%

2.2.1 Exception Handling Implementation

Empirical studies have been conducted to analyze how exception handling is implemented. Cabral and Marques (CABRAL and MARQUES, 2007) performed a study in the context of 16 different programs implemented in Java and of 16 different programs implemented in the .NET platform. These programs were divided into four groups: “Servers”, “Libraries”, “Stand-alone applications” and “Server-apps”. The authors analyzed what types of actions were implemented in exception handlers. For the .NET programs, they observed in the four groups of programs that 72% to 97% of the handlers were categorized as one or more of the following categories: “Empty”, “Log”, “Alternative Configuration”, “Throw” or “Return”. “Empty”, “Log” and “Return” handlers, as their names suggest, are implemented with empty *catch* blocks, logging operations and `return` statements, respectively. Handlers categorized as “Alternative Configuration” perform reconfigurations in the system and those categorized as “Throw” invoke a *throw* statement within the *catch* block to either re-throw or re-map the exception captured by the *catch* block. For the Java programs, the authors also observed in the four groups of programs that 95% of the handlers were in these categories. Table 2.1 summarizes the distribution of handlers in these categories, as reported by Cabral and Marques (CABRAL and MARQUES, 2007). Other handlers analyzed in this study were categorized as: “Assert”, “Close”, “Continue”, “Rollback” and “Others”. Except for the “Assert” handlers, which accounted for approximately 26% of the handlers in the “Libraries” group of .NET programs, these other types of handlers were not very frequent.

One can observe in Table 2.1 that “Log” handlers were common in both .NET and Java programs. One possible explanation for this observation could be the fact that many exceptions raised by third-party libraries do not represent actual errors in the system, so they are usually handled by only logging error messages. Moreover, “Alternative Configuration” handlers were more common in .NET than in Java, whereas “Throw” handlers were more common in Java than in .NET. As Cabral and Marques pointed out, .NET programs favored the implementation of handlers focused on avoiding premature program termination. This may be the reason why most handlers in .NET were categorized as “Alternative Configuration”, “Empty” or “Log”, since these handlers captured exceptions and rapidly allowed the system to continue its normal execution. By avoiding premature program termination with simple handlers, developers provide a minimum level of robustness in their systems. For the Java programs, on the other hand, it was observed that they often centralized system reconfiguration in specific modules implementing “Alternative Configuration” handlers. Then, “Throw” handlers were used to re-map global exceptions, so the re-mapped exceptions can be handled by centralized handlers reconfiguring the system. However, as we shall discuss in the next section, even these simple handlers can violate implicit policies, leading to failures. Finally, it is also interesting to notice that empty handlers were more common in .NET than in Java. One could expect *a priori* that empty handlers were commonly implemented in Java as a way to avoid the reliability checks performed by the Java compiler. It should be noted that programming languages in the .NET platform do not have reliability checks similar to that performed by the Java compiler.

To further investigate how exception handling was implemented in other programming languages, we recently conducted an empirical study (JAKOBUS et al., 2015). First, we analyzed what the most adopted programming languages nowadays are and what kind of exception handling mechanism they provide. By combining a set of indices of programming languages adoption, we produced a list of the 71 most widely adopted programming languages. After examining the specifications of these languages, we concluded that 54 out of the 71 examined languages (approximately 76%) provide built-in exception handling mechanisms. From the 54 programming languages with built-in exception handling mechanisms, 53 follow the *try-catch-throw* model and only 1 – the Go programming language – provides an exception handling mechanism that follows a different model.

Next, we analyzed 50 popular open-source projects hosted on public repositories of the GitHub hosting service, 10 for each of the 5 most adopted programming languages (JavaScript, Java, PHP, C# and C++). These projects

Table 2.2: Size of Exception Handlers

Programming Language	**Empty Handlers**	**Handlers With Only 1 Statement**	**Handlers With More Than 1 Statement**
Javascript	29%	38%	33%
Java	11%	69%	20%
C#	20%	51%	29%
C++	11%	48%	41%
PHP	15%	47%	38%

were analyzed to investigate how exception handlers are commonly implemented in these languages. We investigated the size of the handlers in terms of the number of statements within the *catch* blocks. Table 2.2 depicts the data collected for the size of the handlers. As one can observe in this table, the number of empty *catch* blocks observed in Java and C# projects is very similar to the numbers observed in the study of Cabral and Marques. It is worth mentioning that C# is part of the .NET platform, which was the focus of the study of Cabral and Marques. It is interesting to notice that Java projects had the smallest percentage of empty handlers. Thus, the use of checked exceptions did not seem to influence the existence of empty handlers, as one might expect.

The observations of our study are aligned with those made by Cabral and Marques. For the Java programs, in particular, we observed that few handlers contain more than one statement. These are the handlers that centralized reconfiguring actions, as observed by Cabral and Marques. In addition, we observed that the majority of handlers was simple. These are the handlers that log exceptions that do not represent errors to the system or re-map exceptions so central handlers can handle them, as observed by Cabral and Marques. In fact, implementing simple handlers that delegate to centralized handlers the responsibility of properly handling exceptions seems a reasonable design decision as not all methods in a system are able to implement proper handling actions.

The occurrence of global exceptions is favored by the implicit policies observed, as these policies are often related to propagating exceptions from the place where they occur to modules responsible for actually handling them. In large systems, developers are usually in charge of implementing specific modules. As a consequence, developers might not have the global knowledge required to deal with global exceptions. And as exception handling policies are often only implicitly defined in the source code and not explicitly documented, developers tend to violate them in the source code. As we shall discuss in the next section, most exception-related failures are caused by these violations. Currently, there is still limited support for defining explicit exception handling policies. Exception handling mechanisms in programming languages (Section 2.1.2) are not adequate for that. These mechanisms are intended to implement exception handling, not to design how it should be implemented. Therefore, there is still a need to support the definition of explicit policies in software systems, as well as the detection of violations in the source code. This is further discussed in Chapter 4.

2.2.2 Recurring Problems in Exception Handling

Exception handling is the most common way of dealing with errors during the development of software systems. Indeed, software robustness is improved if exception handling is designed and implemented properly. However, recent empirical studies have shown that the causes of recurring failures in software systems are located in the exception handling code (CACHO et al., 2014a,CACHO et al., 2014b,COELHO et al., 2008,MARINESCU, 2011,MARINESCU, 2013,SAWADPONG et al., 2012).

Sawadpong et al. (SAWADPONG et al., 2012) conducted an exploratory study in the context of six major releases of Eclipse to compute the fault density of the source code. The authors defined fault density as the number of known faults divided by the number of lines of uncommented source code. Then, the authors compared the fault density of the exception handling code to the fault density of the overall code. Their findings showed that the fault density in exception handling code is almost three times higher than the overall fault density. For this reason, the authors claimed that there seems to exist a relationship between the use of exception handling mechanisms and the occurrence of faults in Java programs.

The work of Sawadpong et al. showed that faults are commonly located in the exception handling code, but they did not further investigate how these faults occurred. Marinescu (MARINESCU, 2011) investigated exception handling faults in more details. First, she assessed if Java classes that either raise or handle exceptions are more fault-prone than classes that do not use exceptions. This study showed that classes using exceptions are more likely to exhibit faults than classes not using exceptions. In a subsequent work, Marinescu (MARINESCU, 2013) investigated whether classes that handle exceptions with generic *catch* blocks are more likely to exhibit fault than classes that handle exceptions with specific *catch* blocks. The findings of the study showed that classes that use exceptions and handle them with generic *catch* blocks are more likely to exhibit faults than classes that use exceptions, but do not handle them with generic *catch* blocks.

Other empirical studies have also tried to characterize faults in the exception handling code. Coelho et al. (COELHO et al., 2008) assessed the impact of exception handling in software robustness in the context of aspect-oriented programs. The authors analyzed the robustness of programs implemented in Java and their counterparts implemented in AspectJ. The AspectJ versions of programs were refactored from the Java versions to simplify and better modularize existing concerns, including exception handling. The authors observed that, despite the efforts in simplifying and better modularizing exception handling, the AspectJ programs were less robust than their counterparts in Java. They observed not only a higher number of faults in the AspectJ programs, but also a higher increase in their occurrence during the evolution of the AspectJ programs. These systems were implemented and refactored without relying on explicit exception handling policies, so this may be one of the reasons why exception handling faults were increasingly introduced in the systems along software evolution.

In this study, Coelho et al. also identified patterns of recurring faults located in the exception handling code. These patterns were related to violations of exception handling design decisions. In particular, design decisions defining the places where exception handlers should have been implemented were not implemented as intended in the source code. The authors observed faults related to uncaught exceptions being introduced because aspects did not capture exceptions in the intended places. These aspects were not capturing exceptions in the intended places due to the improper definition of their point-cuts. They also observed that uncaught exceptions were created because aspects explicitly raised them, but no proper *catch* blocks existed in the source code.

To further understand how faults related to uncaught exceptions occur, we performed two empirical studies, one with C# programs (CACHO et al., 2014b) and the other with Java programs (CACHO et al., 2014a). In these studies, we analyzed if and to what extent changes in the normal and exceptional flows of a program introduced faults related to uncaught exceptions. We focused on uncaught exceptions because they usually cause severe failures that make the whole system crash. We performed a change impact analysis and exception flow analysis in the context of 16 evolving C# programs and 16 evolving Java programs. For each group of programs, we identified which change scenarios introduced faults related to uncaught exceptions.

For the C# programs, we observed that two recurring change scenarios accounted for approximately 66% of the exception handling faults. The first recurring change scenario was related to the introduction of *catch* blocks that re-throw the caught exception. This change scenario increased the number of faults because the re-thrown exceptions were not handled in the context of the program. This scenario was responsible for approximately 38% of the exception handling faults. The second recurring change scenario was related to the introduction of *catch* blocks handling exceptions. This change scenario increased the number of faults because one of the statements within the *catch* block raised an exception and this exception was not handled within the program. This scenario was responsible for approximately 28% of the observed faults.

For the Java programs, we observed that three recurring change scenarios accounted for approximately 83% of the exception handling faults. The first change scenario was responsible for almost 48% of the faults related to uncaught exceptions. This change scenario was related to the introduction of *catch* blocks that re-mapped checked exceptions to unchecked exceptions. The number of faults increased because the applications did not handle the unchecked exceptions. The second change scenario was responsible for approximately 22% of the observed faults. This scenario was related to the introduction of *catch* blocks in the source code. The number of faults related to uncaught exceptions increased because one of the statements within the introduced *catch* block raised an exception that was not handled by the program. The third change scenario was responsible for almost 13% of the observed faults and was related to the removal of existing *catch* blocks. This way, exceptions that were previously captured by these *catch* blocks became uncaught.

In these studies, we observed that the change scenarios that introduced faults were similar for the C# and Java programs. They were related to violations of implicit policies. Examples of violations observed were implementing incorrect re-mappings, inappropriately removing *catch* blocks or missing the implementation of *catch* blocks. There is still little support for detecting these violations in the source code, so maybe this is one of the reasons why these violations were increasingly introduced during the evolution of the analyzed programs. Without adequate support, developers can only resort on the features of exception handling mechanisms in programming languages to detect these violations. However, these mechanisms are not intended for supporting the detection of violations. The reliability checks performed by Java, for example, can only detect the lack of existing *catch* blocks for checked exceptions. Even so, developers avoid these checks by re-mapping checked exceptions to unchecked exceptions, which often originated faults in the programs. For this reason, there is still a need to support the detection of exception handling violations in the source code. We further discuss this in Chapter 4.

We also observed in these studies that most faults related to uncaught exceptions were introduced while developers modified the exception handling code. And this was observed in both groups of programs. These modifications in the exception handling code evidence that developers do try to improve it or repair it along software evolution. However, they perform these changes following *ad-hoc* strategies due to the lack of explicit policies in their systems. In some cases, developers were actually trying to repair existing violations in the exception handling code when they introduced others. This finding highlights the difficulty in repairing exception handling violations without causing negative side-effects in other parts of the programs. In these studies, we could observe that developers focused on the local context of the method where the violation was located and overlooked the impact that these modifications might have in other methods due to global exceptions. It was very common to observe developers focusing on capturing and re-mapping exceptions, but forgetting to properly handle the re-mapped exception in other methods, for example. For this reason, there is still a need to support developers while they repair exception handling violations. We further discuss this in Chapter 5.

Finally, the studies presented in this section have focused on quantifying exception handling faults (SAWADPONG et al., 2012, MARINESCU, 2011) or on assessing specific types of exception handling faults, such as those related to generic *catch* blocks (MARINESCU, 2013) and uncaught exceptions (COELHO et al., 2008, CACHO et al., 2014a, CACHO et al., 2014b). Rather than these two kinds of exception handling faults, there is still little empirical knowledge about what other kinds of exception handling faults occur. The ignorance about exception handling faults may contribute to their introduction and hinder their identification in the source code, since developers may not recognize them. In Chapter 3, we present an empirical study we conducted to further investigate what kinds of exception handling faults occur in software systems.

2.3 Support for Exception Handling

Many solutions have been proposed in the last years for assisting developers in dealing with exceptions. These solutions can be clustered in three main groups. Section 2.3.1 presents solutions aiding the comprehension of exception handling code. Section 2.3.2 presents solutions for specifying and verifying constraints that must be satisfied in the exception handling code. Section 2.3.3 presents recommender systems that assist developers in implementing exception handlers.

2.3.1 Exception Handling Comprehension

Software comprehension can be defined as the activity of understanding existing software when developers implement and maintain the source code of software systems (BROOKS, 1983). In the context of exception handling, many solutions have been proposed to support the comprehension of exception handling code (CHANG et al., 2001, FU and RYDER, 2007, ROBILLARD and MURPHY, 1999, ROBILLARD and MURPHY, 2003, SHAH, GÖRG and HARROLD, 2008a).

Robillard and Murphy (ROBILLARD and MURPHY, 1999, ROBILLARD and MURPHY, 2003) investigated the difficulties that developers face when dealing with exceptions during software construction. The authors distinguished between local and global exceptions. Local exceptions are those raised and handled in a single method, whereas global exceptions are those raised and handled in different methods. Robillard and Murphy argued that the greatest difficulty in understanding the exception handling code stem from the impact that global exceptions have in the source code of a program. In this context, they proposed a model to represent the propagation path of global exceptions and implemented the JEX tool to support this model. This tool computes exceptional propagation paths in Java programs and displays these paths as oriented graphs to show which exceptions traverse through specific methods in the program. Thus, the JEX tool supports the process of understanding the propagation of global exceptions in Java programs.

Similarly to Robillard and Murphy, Chang et al. (CHANG et al., 2001) developed a static analysis tool to estimate exceptional propagation paths in Java programs. The authors used these paths to detect unnecessary *try* blocks and *throws* clauses. That is, *try* blocks that guard statements that do not explicitly raise exceptions and *throws* clauses declaring exceptions that do not flow through the boundaries of a given method. Their tool assists developers in simplifying the exception handling code of their programs by detecting unnecessary code that can be removed.

Complementing the works of Robillard and Murphy and of Chang et al., Fu and Ryder (FU and RYDER, 2007) proposed a static analysis technique that considers chained exceptions in Java programs. Chained exceptions occur when *catch* blocks capture and re-throw or re-map exceptions. By considering chained exceptions, the propagation paths of exceptions are analyzed as long paths in which the type of the exception may change, instead of fragmented paths. This technique reduces the number of propagation paths computed by previous techniques, since chained exception are common in Java programs. Given the lower number of propagation paths produced, the comprehension of the propagation of exceptions is facilitated.

Following the same approach of these previous works, Shah et al. (SHAH, GÖRG and HARROLD, 2008a) proposed a visualization tool named ENHANCE – ExceptioN HANdling CEntric visualization. This tool provides three different views for the exception handling code implemented in Java programs. The "quantitative view" displays as a dependency matrix the relationship between packages, classes and methods that raise and handle exceptions. The "flow view" displays as graphs the relationship between methods that raise and handle exceptions. The "contextual view" displays in more details the propagation paths of exceptions, showing not only the places where exceptions are raised and handled, but also the intermediate methods that they traverse.

Although software comprehension is part of the detection and repair of exception handling violations, current solutions supporting exception handling are aimed at only displaying how the exceptions propagate through methods in the source code. At best, these solutions may assist the comprehension of the source code before developers try to detect or repair violations. Visualization-based solutions do not provide visual cues of where violations are located in the source code, nor assists in the comprehension of which modifications must be performed in the source code to repair the violations. Therefore, these solutions do not properly support detecting and repairing exception handling violations.

2.3.2 Specifying and Verifying Exception Handling Properties

Some programming languages provide exception handling mechanisms with extra facilities to further support the construction of robust programs. The Java programming language, for example, allows the specification and verification of exceptional interfaces of methods. In particular, Java verifies at compile time whether client code implements proper *catch* blocks to handle checked exceptions declared in exceptional interfaces. In the exception handling literature, there exist solutions aimed at verifying other exception handling specifications, rather than just those related to exceptional interfaces. In particular, these solutions allow the specification and verification of the places where specific exceptions are expected to be raised and handled. These solutions mainly differ in how these exception handling properties are specified. There are solutions that specify these properties by means of new language constructs added to existing programming languages (CACHO et al., 2008, SILVA and CASTOR, 2013), while other solutions specify these properties with domain-specific languages external to the programming languages (SALES and COELHO, 2011, ABRANTES and COELHO, 2015).

Cacho et al. (CACHO et al., 2008) extended the AspectJ programming language with a new language construct to explicitly define intended properties that the exception handling code must comply with. In particular, the solution proposed by Cacho et al. provides the new construct called "explicit exceptional channel". An explicit exceptional channel defines the types of exceptions that specific methods in the source code should raise and handle. These channels are specified with a new AspectJ point-cut designator and are verified at compile time.

Similarly to Cacho et al., Silva and Castor (SILVA and CASTOR, 2013) extended Java with a new language construct called "exception propagation channel". Conceptually, the exception propagation channels proposed by Silva and Castor are equivalent to the explicit exceptional channels proposed by Cacho et al.. Exception propagation channels also define the exceptions that specific methods are expected to raise and handle.

The solution proposed by Sales and Coelho (SALES and COELHO, 2011) specifies the places where exceptions are expected to be raised and handled using an XML-based idiom. Based on these specifications, their solution generates partial JUnit test cases intended to stimulate the exceptional flow of the system. Then, developers finish the implementation of these test cases and run them to verify whether the implemented system adheres to the specified exceptional contracts. In a subsequent work, Abrantes and Coelho (ABRANTES and COELHO, 2015) extended the work of Sales and Coelho. The solution proposed by Abrantes and Coelho instruments the source code of a program to monitor the adherence of the specifications during program execution. Whenever specifications are violated, notifications are sent to a remote server, where they are stored and later analyzed.

All these solutions rely on the assumption that for designing exception handling it is enough to specify the places where exception raisers and handlers should be implemented. However, the study conducted by Fu and Ryder has already shown that a high number of global exceptions are constituted of chained exceptions (FU and RYDER, 2007), that is, global exceptions are commonly captured and re-thrown or captured and re-mapped along their propagation paths. Moreover, the studies we conducted showed that recurring uncaught exceptions were caused by introducing *catch* blocks that either re-throw or re-map exceptions (CACHO et al., 2014a, CACHO et al., 2014b). None of the aforementioned solutions take into account chained global exceptions, that is, they do not support the specification of the places where global exceptions are supposed to be re-thrown or re-mapped. Therefore, these solutions still provide limited support for the specification and verification of exception handling policies for global exceptions. We further discuss this in Chapter 4.

Even more critical than the limited support provided by these solutions is the lack of support for repairing exception handling violations. The aforementioned solutions can only detect the places where specifications are violated. They do not assist developers in deciding which modifications in the source code must be performed to actually repair these violations. Without this type of support, developers have to repair these violations on their own, at the risk of introducing other violations while they try to repair existing ones. We present our solution for supporting the repair of exception handling violations in Chapter 5.

2.3.3 Exception Handlers Implementation

Recently, recommender systems have been explored as assisting tools for software engineering activities, including development and maintenance activities (ROBILLARD et al., 2010). In the context of exception handling, there exist recommender systems assisting developers in implementing exception handlers (BARBOSA et al., 2012,BARBOSA et al., 2012a,RAHMAN and ROY, 2014).

Barbosa et al. proposed a recommender system to support developers in the implementation of exception handlers in Java programs (BARBOSA et al., 2012, BARBOSA et al., 2012a). Their solution mines open-source projects to build a local repository of code snippets implementing exception handlers. Then, when a developer wishes to handle an exception, he triggers the recommender system to receive recommendations of how they may implement his handler. This recommender system works on the premise that functionally similar methods handle exceptions in similar manners. The recommender system extracts information from the method where the developer wishes to handle the exception to query the repository for similar code snippets. The most similar code snippets are returned to the developer to serve as concrete examples of how he may implement his exception handler. By recommending concrete examples of how exception handlers may be implemented, their recommender system assists developers in the process of discovering handling actions relevant to their context. Thus, developers are expected to implement more effective exception handling actions.

Similarly, Rahman and Roy also proposed a recommender system to support developers in the implementation of exception handlers of Java programs (RAHMAN and ROY, 2014). They also provide code snippets as concrete examples of how exception handlers may be implemented in a given context. However, instead of using a local repository of code snippets, the solution proposed by Rahman and Roy performs its searches in repositories hosted on GitHub. Moreover, it uses different metrics to measure functional similarity in addition to metrics that try to estimates the quality of an exception handler. In particular, their solution estimate the quality of an exception handler by measuring the readability of the exception handler, the average number of statements in each *catch* block of a code snippet and the fraction of the code in the code snippet that is intended for exception handling. This way, their solution is able to recommend more relevant code snippets than the solution proposed by Barbosa et al.

The solutions proposed by Barbosa et al. and by Rahman and Roy are not intended to detect or repair exception handling violations. They are aimed at assisting the implementation of exception handlers. Even if they were employed to support the repair of violations in exception handlers, they would probably not work effectively. The main limitation of these solutions is the fact that they are unaware of the impact of global exceptions. These solutions may provide recommendations that do not introduce faults in the local context of a method where they are applied, but due the impact of global exceptions, these recommendations may introduce violations in other methods of the system. For example, these solutions may recommend re-mapping from an exception to another type, but the re-mapped exception may be left uncaught in the global context of the program. In fact, re-mapping exceptions and leaving them uncaught was commonly observed in the studies we conducted (CACHO et al., 2014a,CACHO et al., 2014b). Therefore, there is still no proper support for assisting the repair of exception handling violations. We present our recommender heuristic strategy that supports the repair of exception handling violations in Chapter 5.

Following a more extreme approach, Cabral and Marques proposed a solution for automatically handling exceptions without intervention of developers (CABRAL and MARQUES, 2008). This solution relies on the assumption that for specific exception types it is possible to define default handlers that can be automatically executed. The runtime system that supports the program execution is responsible for activating these handlers whenever exceptions of pre-defined types are raised. Thus, it is expected that some exceptions be automatically handled by the runtime system without requiring developers to write exception handling code.

However, the assumption that there exist default handling actions for specific exception types does not always hold, since the same exception type can be handled in distinct manners in the same system. The way exceptions are handled vary according to the place where exceptions occur. For example, the way a `FileNotFoundException` should be handled in a operation related to a user choosing a file to open is different from the manner in which it should be handled in a operation that the system is manipulating configuration files. While handling the exception in the first operation would probably require implementing actions to warn the user about the error, the second case would probably require the system to implement reconfiguration actions based on default values, for example. In addition, there are no guarantees that default handling actions will not have any negative side-effects in other parts of the system. Therefore, the solution proposed by Cabral and Marques is only applicable to a restricted set of exception types. In addition, developers would still have to implement the exception handling code for the exceptions defined in their own programs. In this case, the solution proposed by Cabral and Marques would not provide support for detecting and repairing violations in the source code.

2.4 Summary

This chapter presented the main concepts addressed in this thesis. First, Section 2.1 presented the basic terminology adopted throughout this thesis. In particular, it presented basic concepts related to software robustness, such as errors, faults and failures. This section also defined the terminology related to exception handling. Exception handling is the most common way of dealing with errors during software development. A system that fails to properly implement exception handling is likely to continuously crash, compromising software robustness. For this reason, exception handling must be properly designed and implemented in software systems.

Section 2.2 presented previous studies conducted to assess exception handling implementation in software systems. This section presented empirical studies we conducted in collaboration with other researchers, as well as studies conducted by other research groups. The results of our studies in collaboration with other researchers are published in previous papers (CACHO et al., 2014a, CACHO et al., 2014b, JAKOBUS et al., 2015). The studies covered in Section 2.2.1 analyzed how programs implement exception handling.

In particular, these studies observed that programs tend to follow implicit exception handling policies.

The studies presented in Section 2.2.2 showed that the causes of recurring failures in software systems are located in the exception handling code. Despite being consistently reported in software systems, there is still little knowledge about what types of exception handling faults occur. Some studies have only quantified the occurrence of exception handling faults, whereas others have focused on a few kinds of faults, such as those related to uncaught exceptions. Our studies in collaboration with Cacho et al., for example, identified different change scenarios that led to uncaught exceptions. These studies focused on faults related to uncaught exceptions because this type of faults usually cause severe failures that cause system crashes. However, faults related to uncaught exceptions are just one kind of exception handling faults. Other types of exception handling faults may also threaten software robustness without developers even being aware. To bridge this gap, we conducted an empirical study to investigate what types of exception handling faults occur in software systems. This study is further described in Chapter 3.

In the following, Section 2.3 presented existing solutions to support the design and implementation of exception handling. In the exception handling literature, much work has been done on visualization tools that display information about the exception propagation path of exceptions. However, these solutions provide no support for detecting and repairing exception handling violations. Moreover, Section 2.3 also presented solutions aimed at designing exception handling constraints that must be adhered to in the source code. In particular, these solutions rely on the assumption that for designing exception handling is enough to specify the places where exception raisers and handlers should be implemented. However, the studies we conducted in collaboration with Cacho et al. have shown that recurring faults are caused by violations in exception re-mappers and re-throwers. Therefore, one must also design how these re-mappers and re-throwers should be implemented in the source code, as well as detect when the intended implementation is violated. In Chapter 4, we further discuss our solution for supporting the specification and verification of exception handling policies and for detecting exception handling violations in the source code.

Finally, Section 2.3 presented solutions supporting the handling of exceptions. First, we presented solutions based on recommender systems that provide recommendations on how to implement exception handlers. These solutions are not aimed at assisting the detection and repair of exception handling violations. But even if they were employed in the repair of exception handlers, they would fail. They would fail because they are limited to the analysis of local information extracted from a single method, whereas the repair of exception handling violations requires reasoning about the global impact of exceptions. Next, we presented a solution aimed at automatically handling specific exception types with default handling actions. However, the suitability of the handling actions are usually sensitive to the place where exceptions are handled. Therefore, default handling actions are only applicable to a constrained set of exception types. In addition, this solution does not aid the repair of exception handling violations. In fact, there is still no solution aimed at supporting the repair of exception handling violations. In Chapter 5, we present a recommender heuristic strategy that supports the repair of exception handling violations.

3
Investigating Exception Handling Faults

Recent studies presented in the previous chapter showed that most of the causes of recurring failures in software systems are located in the exception handling code (Section 2.2.2). Although the design and implementation of exception handling are intended to improve software robustness, in many cases the exception handling behavior of a system is not able to prevent errors from causing failures. Given the recurrence of failures caused by exception handling faults, developers must be able to detect these faults in their systems. Otherwise, software robustness is compromised.

Currently, there exists tool support for detecting exception handling faults in C programs that use return-code idioms to implement exception handling (BRUTNIK et al., 2006). This tool leverages on a categorization of exception handling faults in return-code idioms commonly adopted in C programs. By knowing what categories of exception handling faults and their characteristics are, the tool is able to statically detect in the source code structural patterns that pinpoint to potential faults. For programs implemented following the *try-catch-throw* model, there is no categorization of exception handling faults. Previous studies have only quantified the occurrences of exception handling faults, or reported faults related to uncaught exceptions or generic handlers (Section 2.2.2). Besides this, little is known about other categories of exception handling faults.

The lack of a categorization of exception handling faults in programs following the *try-catch-throw* model hinders their detection in the source code. Without knowing existing categories of exception handling faults, it is not possible to provide proper support for automatically detecting them in the source code. Therefore, a deeper understanding about what categories of exception handling faults occur in programs is precondition for supporting their detection.

In this context, this chapter presents an empirical study conducted to better understand what categories of exception handling faults occur in programs implemented following the *try-catch-throw* model. We analyzed bug reports in open-source projects and the respective repairing modifications performed in the source code to gather knowledge about exception handling faults. Since little was known about exception handling faults by the time this study was conducted, we could not rely on tools for automatically selecting the bug reports related to exception handling faults. Similarly, we could not rely on tools to automatically extract from version control systems the revisions that repaired exception handling faults.

For this reason, this study relied on a heuristic strategy for identifying the bug reports and the revisions that could be related to exception handling faults. This heuristic strategy required extensive and careful inspections to identify which bug reports and revisions were related to exception handling faults. The thorough inspection during data collection favored the confidence in the collected data, although we are aware that this decision decreased the study scalability. Given the lack of knowledge about exception handling faults, there was no other option other than digging in the data and carefully inspecting it. As the results of this analysis, a categorization of exception handling faults comprising 9 categories emerged from the data. This was the first categorization proposed for exception handling faults in programs following the *try-catch-throw* model.

The investigation about exception handling faults was carried out following the methodology presented in Section 3.1. The analysis of the collected data is presented in Section 3.2 and the results of the study are presented in Section 3.3. The threats to the study validity are discussed in Section 3.4 and related works are presented in Section 3.5. Finally, this chapter is summarized in Section 3.6.

3.1 Settings of the Study

This section describes the settings of the study conducted to investigate exception handling faults. In particular, Section 3.1.1 presents the goal of the study and the questions addressed and Section 3.1.2 details the design of the study.

3.1.1 Goal and Questions

Although existing studies have consistently reported exception handling faults, there is still no categorization of exception handling faults occurring in programs following the *try-catch-throw* model. Without this knowledge, it is not possible to support the detection of exception handling faults in the source code. In this context, the goal of this study is stated as follows:

> ***Goal:*** *Analyze software faults for the purpose of understanding what categories of exception handling faults occur in software systems.*

The exception handling code is the part of a program responsible for signaling and handling the occurrence of exceptions. In mainstream programming languages, the exception handling code is structured with the use of built-in exception handling mechanisms. These mechanisms allow the definition of exception types and the implementation of structural dependencies between methods and exceptions. In particular, the structural dependencies between methods and exceptions comprise the definition of guarded scopes, the definition of terminating actions, as well as the raising, handling, propagation, re-mapping and re-throwing of exceptions (Section 2.1.2). Thus, developers can implement these dependencies to delegate specific exception handling responsibilities to methods in programs. For the sake of brevity, from hereafter the structural dependencies that can be established between methods and exceptions will be called as "exception handling dependencies".

In this context, this study started its investigation by analyzing what exception handling dependencies are implemented in exception handling faults. Therefore, the following question was refined from the previous goal:

> *What exception handling dependencies are implemented in exception handling faults?*

The first question of this study aimed at understanding what exception handling dependencies are implemented in these faults. From hereafter, the exception handling dependencies implemented in faults will be called as "faulty exception handling dependencies" or simply as "faulty dependencies". In addition to this first question, this study also sought to better understand why exception handling faults occur. This investigation was carried out based on basic human errors: errors of commission and errors of omission (SWAIN and GUTTMANN, 1983). Errors of commission occur when results are not within expectations because the actions were executed incorrectly. On the other hand, errors of omission occur when results are not within expectations because actions were not executed. In this study, we employed these concepts to define fault types: faults of commission and faults of omission. If a fault occurs because an exception handling dependency was implemented incorrectly, then this fault is considered a fault of commission. If a fault occurs because an exception handling dependency was not implemented at all, then this fault is considered a fault of omission.

In this context, the following question was also refined from the goal of this study:

Do exception handling faults occur due to commission or omission?

The second question complemented the first one. The first question investigated what faulty exception handling dependencies in programs are, whereas the second question investigated whether these faulty dependencies are typically implemented incorrectly or not implemented at all. By better understanding these characteristics, our initial aim was to lay the foundations for further investigations on the detection and repair of exception handling faults.

Finally, it is worth mentioning that this study investigated exception handling faults focusing on their observable features. We did not focus in assessing the adequacy of the underlying intentions and beliefs that motivated developers during software construction. This discussion is beyond the scope of this study.

3.1.2 Study Design

To achieve the goal and answer the questions defined in Section 3.1.1, this study analyzed faults reported in software projects. To collect the data required for this study, the target systems selected must meet the following criteria. First, the target systems must have an active process of reporting and registering the detected faults, so that the descriptions associated to the faults can be analyzed. Second, the target system must have the history of modifications in their source code registered, so that the repairing modifications can be analyzed. Third, it must be possible to map a given bug report to the respective repairing modifications, so that the descriptions in the bug report can be related to the respective modifications in the source code. Last, the bug reports and the history of modifications in the source code must be publicly available, so that they can be analyzed.

Given these criteria, open source projects were good candidates for target systems of this study since they provide their source code in publicly available version control systems. In addition, these projects report the detected failures in publicly available bug reports. In fact, open source projects typically have an active community of users that proactively participate in the development process by asking for new features and reporting failures (ZHAO and

ELBAUM, 2003). These projects usually have multiple developers spread all over the world contributing to the source code (ZHAO and ELBAUM, 2003). For this reason, their issue tracking systems are an important means of communication between developers. Moreover, open source projects usually have quality assurance developers to triage and confirm the reported failures (ZHAO and ELBAUM, 2003). Thus, there is a satisfactory degree of confidence that the reports that are confirmed by the quality assurance team of open source projects are actual failures. Consequently, the issue tracking systems of open-source projects are a valuable source of information about the existing faults in these projects.

Procedure for Data Collection

The data required for this study consisted of bug reports related to exception handling faults and the commits that repaired these faults. To link bug reports to their respective commits, the heuristic strategy described by Bachmann and Bernstein was employed (BACHMANN and BERNSTEIN, 2009). This strategy relies on the observation that many software projects adopt patterns in their commit messages. In particular, these projects demand that their developers put as part of their commit messages an identifier in a standard format specifying to which bug report that commit is related. This standard format is project-dependent. It can be the URL of the bug report, numbers in combination with specific keywords (e.g. "Fixes #10", "BugId-17"), among others. The strategy described by Bachman and Bernstein works by scanning through the commit messages searching for a given pattern. Then, the commits whose messages match a given pattern are manually inspected to discard false positives. First, we verified whether the matched pattern actually refers to a valid bug report in the issue-tracking system. Then, we also verified whether the referred report relates to a fault, or to another kind of issue (e.g. "New Feature"). In this study, the heuristic strategy described by Bachmann and Bernstein was adapted to search not only for the project-dependent commit message pattern, but also for the keyword "exception". We opted for this strict criterion to improve the chances and confidence that commits were indeed related to exception handling.

The data collection procedure of this study followed the workflow depicted in Figure 3.1. (i) Commits whose messages matched the project-dependent commit message pattern and the keyword "exception" are called as "commits potentially related to exception handling faults". Then, the commits potentially related to exception handling faults were manually inspected to discard false positives. (ii) First, we checked whether the pattern matched actually re-

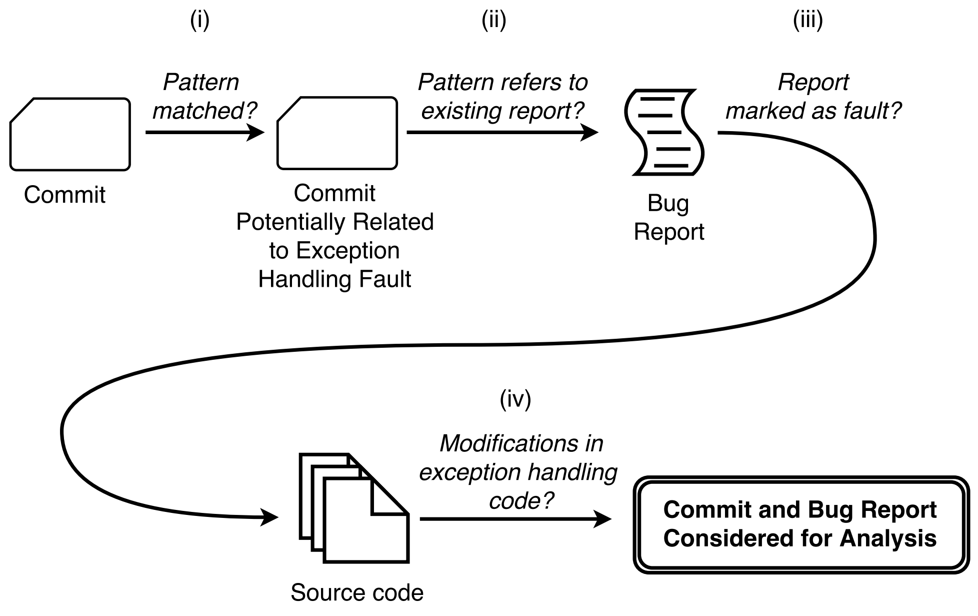

(i)
Pattern matched?
Commit
(ii)
Pattern refers to existing report?
Commit Potentially Related to Exception Handling Fault
(iii)
Report marked as fault?
Bug Report
(iv)
Modifications in exception handling code?
Source code
Commit and Bug Report Considered for Analysis

Figure 3.1: Data Collection Procedure

ferred to an existing bug report in the repository. For those that referred to an existing report, the respective report was inspected to confirm whether it was related to a failure or not. (iii) If a given report was not marked as a failure by the development team, then their opinion was considered as sovereign and the given commit and the respective report were not considered for analysis. If the report was marked as being related to a failure, then we manually reviewed the corresponding changes performed in the source code. (iv) If the modifications performed in the source code were related to the definition of guarded scopes, the definition of terminating actions, the declaration, raising, handling, propagation, re-mapping or re-throwing of exceptions, then the fault was considered an exception handling fault.

Finally, it is worth noting that due to the little knowledge about exception handling faults in programs following the *try-catch-throw* model, this study had an inherent exploratory nature. Without *a priori* knowledge about exception handling faults, it was not possible to build tools for assisting the automatic identification of bug reports and commits related to these faults. For this reason, the data collection procedure of this study relied on extensive and careful inspections to certify that the bug reports and commits identified were related to exception handling faults. Thus, the thorough inspections favored the confidence in the collected data, at the cost of reducing the scalability of the data collection procedure.

Procedure for Data Analysis

To analyze the collected data, for each exception handling fault collected, we reviewed the corresponding repairing modifications performed in the source code. To support the review of the changes performed in the source code, the SemDiff tool was used (DAGENAIS and ROBILLARD, 2011). The SemDiff tool downloads to a local repository all commits available on a given version-control system and computes the diff between each pair of subsequent revisions. During the source code review, the observed modifications performed between the faulty and the repaired versions were textually described. For example, by analyzing the modifications performed to repair a fault analyzed in this study, the textual description produced was:

Changed the type of the exception used by a throw statement.

As one can observe in the previous description, the fault was repaired by modifying the type of the exception raised by an existing *throw* statement. After describing the modifications in the source code, we analyzed the faults according to the following criteria. If the fault was repaired by adding new exception handling dependency, then it was considered that the fault occurred due to the lack of this dependency. Therefore, this fault was considered a fault of omission. On the other hand, if the fault was repaired by modifying or removing an existing exception handling dependency, then it was considered that the fault occurred because the existing dependency was incorrect. Therefore, this fault was considered a fault of commission.

Besides reviewing the source code, we also reviewed the textual description and comments available in the bug reports, as well as the commit messages. From these data sources, we extracted sentences describing the faults or explaining how they should be repaired. For the aforementioned fault, the following comment was extracted from its bug report:

(...) Tomcat is however not throwing the right exception in this case

As one can observe in the previous comment, the problem associated to this fault was an exception being raised with a type different from what was expected. Thus, the sentences extracted from bug reports and commit messages were used to support the observations of the modifications in the source code, supporting the identification of what exception handling dependencies were implemented and whether they were incorrectly implemented, or not implemented at all.

Finally, the faults were categorized as follows. Faults of omission were categorized as "Missing <*dependency*>" faults, where <*dependency*> refers to the name of the exception handling dependency observed in the fault (e.g., “Missing re-mapper” or “Missing handler”). Similarly, faults of commission were categorized as “Incorrect <*dependency*>” faults.

For the aforementioned fault, we observed that modifying an existing raiser repaired it. More specifically, modifying the exception used by an existing *throw* statement repaired the fault. Moreover, based on the comment extracted from the bug report and the repairing modifications performed in the source code, we could understand that this fault occurred due to an incorrect implementation of a raiser. In other words, this fault occurred due to a fault of commission. Therefore, this fault was categorized as “Incorrect raiser”.

Target Systems

A set of open-source projects that adopt a commit message pattern was identified. Most of the projects of the Apache Software Foundation, such as Ant, Hadoop, Ivy, Maven, Tomcat, among others, adopt project-specific commit message patterns. Among the projects from the Apache Software Foundation, we selected as the target systems of our study those with strict robustness requirements: the Tomcat web server and the Hadoop framework. Both Tomcat and Hadoop are two of the most complex and adopted open source projects nowadays. Moreover, they have large and active communities of users and developers.

Among the list of potential candidates from the Apache Software Foundation, Tomcat is the longest-lived project and is widely adopted in large-scale production environments due to its maturity and robustness. On the other hand, Hadoop is a younger project that has rapidly gained fame in industrial settings in the last years due to its performance and robustness. With these two target systems, we expected to observe source code produced by development teams concerned with the quality of the exception handling, due to the robustness requirements of their systems. This would allow us to observe how exception handling faults occur and how they are repaired in the context of robust, complex and large-scale software systems that have successfully evolved over the years.

By the time this study was conducted, the Apache Software Foundation maintained two main development branches for Tomcat, versions 6.0.x and 7.0.x, and a single development branch for Hadoop. Only these two versions of the Tomcat had the complete history of evolution of the source code available on an open version-control system. Therefore, we analyzed the complete revision history of the versions 6.0.x and 7.0.x available on the Tomcat version-control system and the complete revision history of the main development branch of Hadoop as collected on April 25, 2013.

3.2 Data Analysis

This section presents the analysis of the data gathered in this study. Section 3.2.1 gives an overview of the collected commits and Section 3.2.2 presents the categories of exception handling faults identified in this study.

3.2.1 Collected Commits

The data collection procedure of this study identified 63 commits potentially related to exception handling faults in the context of Tomcat and 105 commits potentially related to exception handling faults in Hadoop. From this total of 168 commits, a tally of 54 commits were identified as being related to exception handling faults, which accounts for approximately 32% of the total. The rest of the commits were not considered for analysis for different reasons. The chart shown in Figure 3.2 summarizes the reasons why these commits were not analyzed.

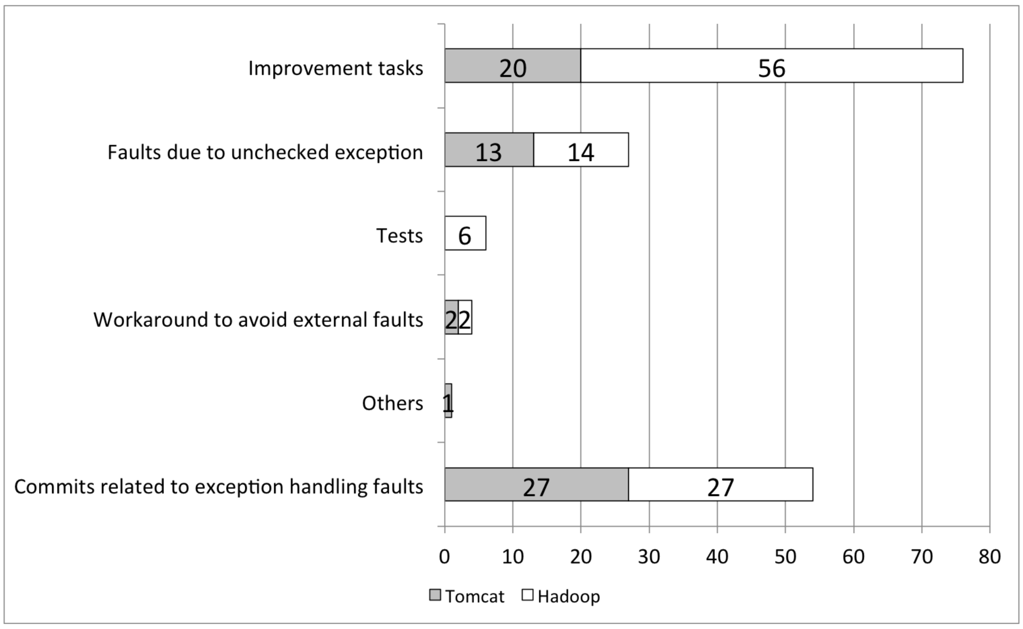


Figure 3.2: Collected Commits

The first category ("Improvement tasks") encompasses commits that were related to bug reports marked as improvements to existing features of the system. Examples of commit messages in this category are: *"Changed exception type thrown when session manager exceeds active session limit"*, or

"Trim long exception messages". Thus, these commits were somehow related to exceptions, but not to faults. It is worth noticing that almost 45% of the collected commits were actually related to improvements in exception handling code. This suggests that during the evolution of the target systems the development teams were continually addressing issues related to improvements in the exception handling code of the systems. This way, our initial assumption that projects with strict robustness requirements would be more concerned with the proper implementation of exception handling seemed to hold for our target systems.

The second category ("Faults due to unchecked exception") encompasses commits that repaired faults related to the occurrence of unchecked exceptions. As discussed in Chapter 2, in the context of the Java programming language, unchecked exceptions are exceptions that typically represent infringements to constraints of the language semantics. Therefore, commits related to unchecked exceptions infringing constraints of the language semantics were not considered related to exception handling faults, since the root cause of the problem was not related to exception handling.

The third category ("Tests") encompasses commits related to the implementation of tests. Next, it is shown an example of a commit message in this category:

> *Test org.apache.hadoop.fs.TestFilterFileSystem fails due to java.lang.NoSuchMethodException.*

This way, these commits were discarded because they were not related to exception handling faults, which are the target of this study.

The fourth category ("Workaround to avoid external faults") encompasses commits that implemented workarounds to avoid the side effects of faults in third party libraries. One of the commits of this category, for instance, had the following message:

> *Avoid ArrayIndexOutOfBoundsException triggered by Java 6/7 XML parser bug.*

As one can observe in the previous message, the root cause of the problem was a bug in a third-party library. For this reason, this commit was not considered for analysis.

The fifth category ("Others") encompasses only one commit that repaired a fault due to an infinite loop that was implemented within an exception class. This commit had the following message:

Prevents infinite loops when an exception is thrown the returns itself for getRootCause().

This commit was also not analyzed because it was a fault not related to an exception handling dependency, but to an incorrect instruction in the normal code.

Finally, the last category presented in Figure 3.2 (“Commits related to exception handling faults”) encompasses the commits that were related to exception handling faults. A total of 27 commits from Hadoop and 27 commits from Tomcat were identified as being related to exception handling faults.

3.2.2 Categories of Exception Handling Faults

After the triage process, a total of 54 commits were identified as being related to exception handling faults. It is worth mentioning that 8 of these commits were related to two exception handling faults and 1 commit was related to three different exception handling faults. The other commits were related to only one exception handling fault each. Therefore, even though the number of commits related to exception handling faults is 54, the total number of exception handling faults observed in this study is 64. From this total, there were 35 exception handling faults in Hadoop and 29 faults in Tomcat.

The collected data for the categories of exception handling faults is presented in Table 3.1. Each row in Table 3.1 corresponds to the faulty exception handling dependencies observed in this study. The columns labeled as “Faults of Commission” and “Faults of Omission” refer to the fault types considered in the data analysis procedure, as described in Section 3.1.2. The last row presents the sum of faults per fault type and the last column presents the sum of faults per exception handling dependency.

Table 3.1: Collected Data for Exception Handling Faults

	Faults of Commission	**Faults of Omission**	**Total**
Handler	20	6	26
Raiser	5	15	20
Re-mapper	9	3	12
Terminating Action	3	2	5
Guarded scope	1	0	1
Total	38	26	64

As one can observe in Table 3.1, faults of commission were more frequent than faults of omission: approximately 59% of the faults were faults of commission. Moreover, we observed 5 exception handling dependencies in the analyzed faults. The three most frequent dependencies – "Handler", "Raiser" and "Re-mapper" – appeared in approximately 90% of the faults analyzed in this study. The other dependencies observed in this study were "Terminating actions" and "Guarded scope".

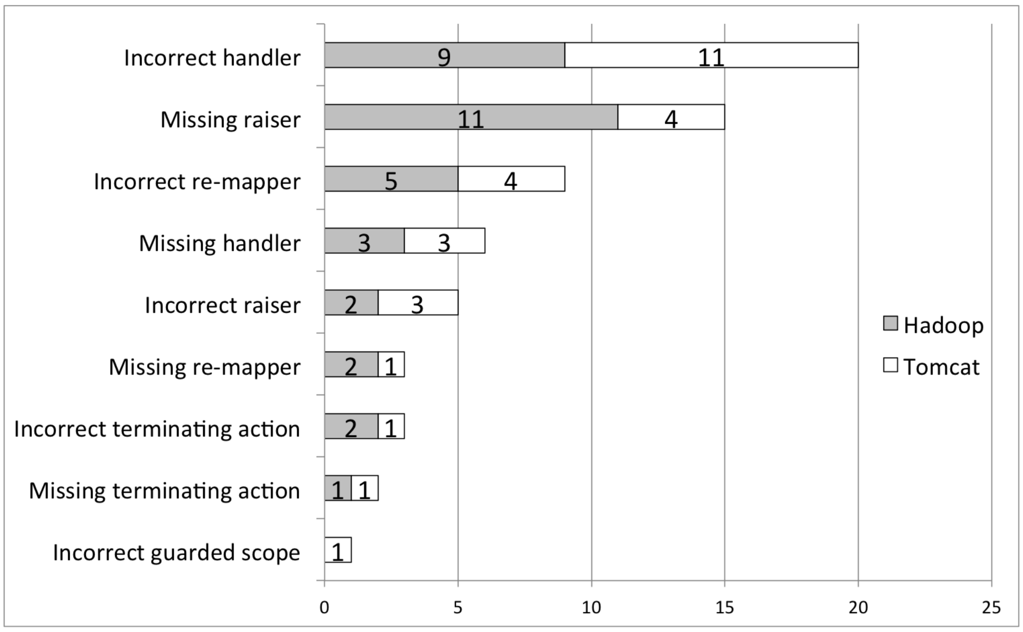


Figure 3.3: Categories of Exception Handling Faults

Based on the analysis procedure presented in Section 3.1.2, we identified 9 categories of exception handling faults. These categories are presented in the chart depicted in Figure 3.3. Among the categories identified, the most frequent categories were: "Incorrect handler", "Missing raiser" and "Incorrect re-mapper". These categories sum a total of 44 faults, which is approximately 70% of the total faults analyzed. The other categories identified were: "Missing handler", "Incorrect raiser", "Missing re-mapper", "Incorrect terminating action", "Missing terminating action" and "Incorrect guarded scope". In the next sections, each category is detailed.

Incorrect Handler

There were 20 faults classified as "Incorrect handler". These were the most frequent faults observed in this study, comprising approximately 31% of the analyzed faults. Exception handling faults in the "Incorrect handler" category occur when an exception handler is implemented in the source code, but it is implemented incorrectly. Different reasons of why exception handlers were incorrect were observed. These differences were also analyzed in order to identify sub-categories of the “Incorrect handler” category. The chart depicted in Figure 3.4 presents the sub-categories of faults identified in the context of the “Incorrect handler” category.

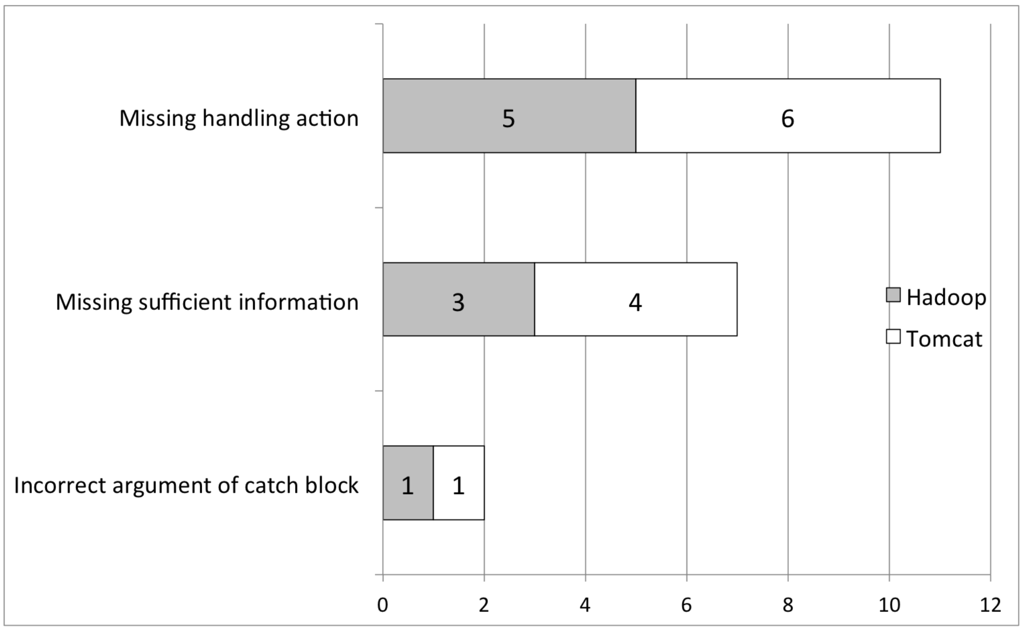


Figure 3.4: Sub-Categories of Incorrect Handlers

One can observe in the chart depicted in Figure 3.4 that 3 different sub-categories of faults were observed in the context of the “Incorrect handler” category. The identified sub-categories were: “Missing handling action” (11 faults), “Missing sufficient information” (7 faults) and “Incorrect argument of catch block” (2 faults). Next, each one of these sub-categories is detailed.

Missing Handling Action. Faults in the “Missing handling action” sub-category are related to the lack of specific actions within a *catch* block. In particular, we observed the lack of handling actions responsible for reconfiguring the system, the lack of handling actions responsible for releasing pre-allocated resources and the complete lack of handling actions for specific exceptions.

There were 4 faults related to the lack of specific actions responsible for reconfiguring the system in an attempt to restore it to a correct state. Without these reconfiguration actions, after the *catch* block is executed, the system continued its normal flow in an incorrect state, which ultimately led to failures.

There were 4 faults related to pre-allocated resources that were expected to be released in case of exceptions, but no specific actions for releasing them were implemented in an existing *catch* block. When exceptions occurred in the *try* block, the resources were not released in *catch* blocks, nor in *finally* blocks, which ultimately led to failures caused by resource leaks.

Finally, there were 3 faults related to *catch* blocks capturing exceptions and not implementing any handling action for them. These *catch* blocks ignored the captured exceptions, when they should not be ignored. Two out of the three faults in this sub-category were related to completely empty *catch* blocks. The other fault in this category, which was observed in Tomcat, ignored the caught exception without using an empty *catch* block. The following code snippet adapted from the source code of Tomcat depicts this fault:

```
try{ S }
catch( GenericException e ) {
   if( e instanceof SpecificException ){
      // handle exception
   }
}
```

In the previous code snippet, the *catch* block declares a generic exception type as its argument, but it uses an *if* statement to select which exception types it will actually handle. There are no handling actions for the exceptions captured by the *catch* block that do not match the condition specified in the *if* statement. Therefore, these exceptions were silently ignored by the *catch* block, when they should not be.

Missing Sufficient Information. Faults in the “Missing sufficient information” sub-category are related to exception handlers not providing important information related to the exception to other parts of the system. The following sentences extracted from bug reports exemplify descriptions of the faults in this sub-category:

> *The getPassword() method of the DataSourceRealm does not log enough information when it encounters an SQL exception.*

> *(The catch block) does not provide any information on the file where it (the exception) occurred.*

The previous sentences exemplify faults that occurred because handlers did not log sufficient information about exceptions. It should be noted that both Hadoop and Tomcat are systems whose users are developers. Thus, it is important that these systems provide sufficient information about exception occurrences to developers, so they can monitor the execution of the system.

Incorrect Argument of Catch Block. Faults in the "Incorrect Argument of Catch Block" sub-category are related to *catch* blocks declaring incorrect exception types as their arguments. In particular, the 2 analyzed faults classified in this sub-category were related to *catch* blocks declaring overly-generic exception types, which incorrectly captured exceptions by subsumption (Section 2.1.2). The problems related to these faults occurred because these exceptions should not be captured where these *catch* blocks were implemented. This way, exceptions that were expected to flow out of a given method to be handled somewhere else were unintentionally captured by subsumption by *catch* blocks declaring overly-generic exception types.

Missing Raiser

There were 20 faults classified as "Missing raiser". Faults classified in this category were the second most frequent faults, comprising approximately 23% of the analyzed faults. Faults in this category are related to exceptions not being raised in conditions that they should be. In this study, all faults in this category were related to the lack of *throw* statements in the source code to raise exceptions when erroneous conditions were met. We observed "Missing raiser" faults caused by the lack of checking conditions to detect erroneous conditions, allowing the systems to continue their execution in inconsistent states, which ultimately led to failures. We also observed faults in this category caused by erroneous conditions being signaled using the return of error codes, when exceptions were expected. The following example depicts one "Missing raiser" fault caused by the lack of checking conditions observed in Tomcat:

```
// SSL protocol
int value = SSL.SSL_PROTOCOL_ALL;
if ("SSLv2".equalsIgnoreCase(SSLProtocol)) {
   value = SSL.SSL_PROTOCOL_SSLV2;
} else if ("SSLv3".equalsIgnoreCase(SSLProtocol)) {
   value = SSL.SSL_PROTOCOL_SSLV3;
} else if ("TLSv1".equalsIgnoreCase(SSLProtocol)) {
   value = SSL.SSL_PROTOCOL_TLSV1;
} else if ("SSLv2+SSLv3".equalsIgnoreCase(SSLProtocol)) {
   value = SSL.SSL_PROTOCOL_SSLV2 | SSL.SSL_PROTOCOL_SSLV3;
}
// Create SSL Context
(...)
```

The previous code snippet depicts the portion of code of a method in Tomcat that initializes a SSL connection (Secure Sockets Layer). As one can observe in the code, the `value` variable is initialized with the default value `SSL.SSL_PROTOCOL_ ALL`. Next, the `if-else-if` statements check what the configuration of the protocol to be used in the connection is, which is stored in the `SSLProtocol` variable. In case no `if-else-if` matches the expected values for this variable, the system continues its execution with the variable `value` containing its initial value. Allowing the system to continue its execution with this default value in case of a missing configuration was the cause of the problem, as pinpointed by a comment in the bug report:

> *Misconfiguration of an SSLProtocol should never silently fall back to enabling all protocols. At minimum, misconfiguration of this value should result in error messages. Since SSLv2 is vulnerable to several attacks known to have some serious security flaws even allowing the possibility of man-in-the-middle attacks, I think a misconfiguration should cause the connector to fail.*

As pointed out in the previous comment, the fault was caused because a misconfiguration in the system allowed it to continue its execution with more privileges than it should, possibly causing a security flaw in the system. This fault was repaired by adding an `else` statement to detect the cases of misconfigurations and raise exceptions to prevent the system to establish the connection.

We also observed “Missing raiser” faults caused because error conditions were signaled returning error codes, when exceptions were expected. The following comment exemplifies these cases:

> *Currently the renew and cancel delegation token method will return false if something goes wrong, but we need an exception.*

As pinpointed in the previous comment, the methods returned “false” in erroneous conditions instead of raising exceptions. This fault, in particular, was repaired by replacing the *return* statement by a *throw* statement.

Incorrect Re-mapper

Faults classified as “Incorrect re-mapper” were the third most frequent faults observed in this study. There were 9 faults in this category, comprising approximately 14% of the analyzed faults. Faults in this category are related to exceptions being re-mapped to other exception types in incorrect manners. In particular, we observed two distinct cases in which exceptions were incorrectly re-mapped: “Incorrect wrapping” and “Re-mapping between incorrect types”.

Incorrect Wrapping. Faults in the sub-category "Incorrect wrapping" are related to the re-mapped exceptions being instantiated with incorrect or insufficient information. This sub-category comprised 5 faults. The following comment extracted from a bug report of Tomcat exemplifies the problem related to this cause of incorrect re-mapping:

> *(The fault is) the fact that the catch block is swallowing the original exception and (its) stack trace.*

The following code snippet exemplifies the problem described in the previous comment:

```
catch(T1 e){
   throw new T2( e.getMessage() );
}
```

In the previous code snippet, the new exception is instantiated using only the message of the original exception. This instantiation loses all the other information contained in the captured exception. It is worth mentioning that this practice is not always a fault, since in some cases we actually do not want to expose information about the original exception due to the information hiding principle.

Re-mapping Between Incorrect Types. Faults in the sub-category "Re-mapping between incorrect types" are related to exceptions being re-mapped from one type to another, when at least one of the two types is incorrect. Faults in this sub-category may occur because a *catch* block re-mapping exceptions captures the incorrect exception type, or because the *throw* statement re-mapping exceptions raises the incorrect exception type. In this study, 1 fault in this sub-category was related to the first case and 3 faults were related to the second case.

Missing Handler

Faults classified as "Missing handler" were the fourth most frequent faults observed in this study. There were 6 faults in this category, comprising approximately 9% of the analyzed faults. Faults in this category are related to the absence of *catch* blocks in the source code to handle specific exceptions. In this study, 5 out of the 6 cases classified as "Missing handler" caused the termination of the program execution due to uncaught exceptions. A tally of 3 out of these 5 faults that caused program termination were caused by faults categorized as "Re-mapping Between Incorrect Types". In particular, "Re-mapping Between Incorrect Types" faults re-mapped the caught exceptions to the `RuntimeException` type and no handler existed to handle the re-mapped exceptions. This way, the re-mapped caused system crashes.

The other "Missing handler" fault, which was observed in Hadoop, did not cause a program termination, but caused an incorrect behavior of the system due to the premature termination of an operation. This fault, in particular, was related to a method terminating its execution without retrying, when it was expected that this method would retry this operation before terminating its execution. The following code snippet depicts the repaired version of this fault:

```
try {
   result = in.read();
} catch (IOException e) {
   LOG.info("Attempting to reopen " + key);
   seek(pos);
   result = in.read();
}
```

In the previous code snippet, the `in.read()` method invocation is guarded by a *try-catch* statement, so that `IOException` is handled. In the faulty version of the code, this method invocation was not guarded and the `IOException` propagated to callee methods. The fault was repaired catching the exception and retrying the operation that initially raised the exception. To do this, the handling actions implemented within this *catch* block logged an error message, invoked the `seek(pos)` method to reposition the file-pointer in the input stream and then retried invoking the `in.read()` method. If this method invocation raised an `IOException` again, then this exception was propagated to callee methods.

Incorrect Raiser

The "Incorrect raiser" category comprised 5 faults, accounting for approximately 8% of the observed faults. Faults in this category are related to exceptions being raised incorrectly. In particular, three causes of exceptions being incorrectly raised were observed: "Incorrect type raised", "Incorrect raising condition" and "Incorrect exception declaration".

Incorrect Type Raised. A tally of 3 faults were categorized as "Incorrect type raised". Faults in this sub-category are related to *throw* statements raising exceptions in correct exceptional conditions, but raising the exceptions with the incorrect exception type. The following comment exemplifies faults in this sub-category:

(...) Tomcat is however not throwing the right exception in this case

As pointed out in the previous comment, the fault was caused because a given module in Tomcat was not raising exceptions with the correct type. This fault was repaired by modifying the type of the exception raised.

Incorrect Raising Condition. Only one fault was categorized as "Incorrect Raising Condition". This fault is related to an exception being raised in a condition in which it should not be raised. In particular, the exception was raised in a non-erroneous situation because the condition of an *if* statement was incorrectly defined. This fault was observed in Tomcat.

Incorrect Exception Declaration There was only one fault classified as "Incorrect exception declaration". This fault was observed in Tomcat and is related to an exception being declared as a static final field in a class. This fault was related to a memory leak, which was actually caused by a problem in the class loader of the Java virtual machine. A comment extracted from the respective bug report of the Java virtual machine pinpoints to the symptom of this fault:[1]

This frequently occurs in servlet containers and can result in a significant memory leak on web application reload.

The following comment was extracted from the Tomcat bug report and pinpoints to the cause of the problem observed in this system:

This issue (memory leak) is apparently caused by an instance of EnableDTDValidationException that is being kept in a static final field.

Then, the following comment also extracted from the Tomcat bug report proposed a solution for repairing this fault:

To avoid it in Tomcat, I vote for "Do not cache the Exception. Create a new instance each time".

In summary, all the raisers of a class were using an exception declared as a static field. This exception was declared as a static field in the class to avoid creating a new exception object every time an exception had to be raised by the methods of a class. However, declaring the exception as a static field

[1]`http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6916498`

caused a memory leak in Tomcat due to a problem in the class loader of the Java virtual machine. As explained in the previous comment, this fault was repaired by not caching the exception. Thus, the exception declaration in a static field was removed and all *throw* statements of this class were modified to raise a new exception object.

Missing Re-mapper

The “Missing re-mapper” fault category comprised 3 faults, accounting for approximately 5% of the observed faults. Faults in this category are related to exceptions being propagated to out of methods without being re-mapped, when they were expected to be re-mapped to different types. These faults occurred because the exceptions that propagated to out of the methods were not expected in other modules of the system. One of the comments in a bug report explained:

> *(The module) DF should use a more reasonable exception when mount cannot be determined.*

As pinpointed in the previous comment, the problem occurred because the DF module was not using a proper exception. In this fault in particular, the module was letting exceptions flow out of its bounds without being re-mapped. The following simplified code snippet depicts the repaired version of this fault:

```
 try {
   (...)
   this.mount = tokens.nextToken();
   (...)
   } catch (NoSuchElementException e) {
      throw new IOException("Could not parse line: " + line);
   } catch (NumberFormatException e) {
      throw new IOException("Could not parse line: " + line);
   }
}
```

In the previous code snippet, one can observe that inserting re-mappings from `NoSuchElementException` to `IOException` and from `NumberFormatException` to `IOException` repaired the fault. In the faulty version of this code, these re-mappings did not exist. Thus, the method allowed `NoSuchElementException` and `NumberFormatException`. And callee modules did not expect these exceptions.

Incorrect Terminating Action

The "Incorrect terminating action" fault category comprised 3 faults, accounting for approximately 5% of the observed faults. Faults in this category are related to terminating actions implemented in *finally* blocks being incorrectly implemented. In particular, all faults in this category were related to actions within the *finally* unintentionally suppressing exceptions raised or propagated from the statements guarded by *try* blocks. The following simplified code snippet adapted from Hadoop exemplifies faults in this category:

```
public void copyBytes()throws IOException{
   try {
      openStream();
      // manipulate and copy bytes
   } finally { closeStream(); }
}
```

In the previous code snippet, assume that an `IOException` is raised from one of the statements guarded by the *try* block. Then, when the *finally* block is executed, it also raises an `IOException`, which suppresses the first exception raised in the *try* block. This way, the exception raised or propagated from the *try* block is lost.

Missing Terminating Action

The "Missing terminating action" fault category comprised 2 faults, accounting for 3% of the observed faults. These faults are related to the absence of *finally* blocks in the source code. In particular, faults in this category are related to the lack of terminating actions responsible for releasing pre-allocated resources. This category is similar to some of the cases of the sub-category "Missing handling action". The difference between these two categories of faults is the location of where the terminating actions were expected to be implemented. In the "Missing terminating action" category, the actions for releasing resources were expected to be implemented in *finally* blocks. It was expected that the pre-allocated resources were released in both normal and exceptional termination of the methods. In the "Missing handling action", on the other hand, the actions for releasing resources were expected to be implemented only in *catch* blocks. It was expected that the pre-allocated resources were released only when the methods terminate their execution exceptionally. This slight detail differentiates these two categories.

Incorrect Guarded Scope

There was only one fault in the "Incorrect guarded scope" category. An overly-protective *try* block caused the fault observed in this category. The *try* block was so long that it guarded more statements than it should. The following comment extracted from the respective bug report pinpoints to this problem:

> *I believe the scope for which the try-catch FileNotFoundException' block applies is too great.*

The long *try* block observed in this fault caused some guarded statements to be skipped in case of exceptions, when these statements should not be skipped. When exceptions occurred in one of the first statements of the long *try* block, a transfer of execution control to the first matching *catch* block occurred and the statements in the middle and in the final of the *try* block were not executed.

3.3 Results and Discussions

This section presents the results of this study, as well as the discussion of their implications. In particular, Section 3.3.1 presents the results that address the first and the second questions of this study. Then, Sections 3.3.2, 3.3.3 and 3.3.4 present other findings of this study.

3.3.1 Exception Handling Dependencies and Fault Types

The analysis of the collected exception handling faults supports the answer to the questions of this study:

> *What exception handling dependencies are implemented in exception handling faults?*
>
> *Do exception handling faults occur due to commission or omission?*

In this study, we observed 5 exception handling dependencies in the analyzed faults. In particular, the exception handling dependencies observed in this study were: "Handler" (26 faults), "Raiser" (20 faults), "Re-mapper" (12 faults), "Terminating action" (5 faults) and "Guarded scope" (1 fault). The 3 most frequent dependencies – "Handler", "Raiser" and "Re-mapper" – were observed in almost 90% of the analyzed faults. In addition, exception handling faults occurred due to commission and omission, but faults of commission were a bit more frequent than faults of omission. From the 64 exception handling faults analyzed in this study, 38 faults were faults of commission, which is approximately 59% of the total. Faults of commission were further classified as: “Incorrect handler” (20 faults), ”Incorrect re-mapper” (9 faults), “Incorrect raise” (4 faults), “Incorrect terminating action” (3 faults) and “Incorrect guarded scope” (1 fault).

It is interesting to notice that the exception handling dependencies “Handler” and “Re-mapper” were the first and second most frequent and they are both implemented with *catch* blocks. Thus, faults in *catch* block comprised approximately 59% of the total faults. Also, faults classified as “Incorrect handler” and ”Incorrect re-mapper” comprise approximately 76% of the faults of commission. In other words, approximately 76% of the faults of commission were observed in *catch* blocks. Therefore, exception handling faults are frequently observed in *catch* blocks and these *catch* blocks are often implemented incorrectly. A possible explanation to this could be the fact that Java performs at compile-time automatic reliability verification for the checked exceptions. This verification requires that checked exceptions are either captured by *catch* blocks or declared in *throws* clauses. This way, it is likely that in the early versions of the system developers implement simple *catch* blocks to avoid compilation errors signaled by the Java compiler. Also, they tend to implement simple *catch* blocks in early versions, when features are not completely implemented and the proper exception handling may not be fully understood. Then, as software evolves and core features are fully implemented, it becomes clear how exceptions should be handled. Thus, developers try to improve these simple *catch* blocks. As presented in Section 3.2.1, there were many commits related to improvements in exception handling. However, some of these simple *catch* blocks may remain unmodified in the source code, leading to failures in later versions of the system.

The 26 faults of omission were further classified as: “Missing raiser” (15 faults), “Missing handler” (6 faults), “Missing re-mapper” (3 faults) and “Missing terminating action” (2 faults). A possible explanation to the higher frequency of “Missing raiser” could be the fact that these faults represent exceptional conditions that are only discovered along software evolution. These new exceptional conditions may arise due to new features being implemented in the system, or existing features that are better understood as they are more exercised along software evolution. This might be the reason why “Missing raiser” faults were more frequent in Hadoop (11 faults) than in Tomcat (4 faults). In fact, this was the only fault category in which a big difference between the two target systems was observed. Since Hadoop is a younger project than Tomcat, it still had many new features being incorporated to its core functionalities, as well as existing features that were still maturing. Thus, it might be the case that exceptional conditions were still not fully understood by Hadoop's development team in the early versions of the system and were only discovered and incorporated along software evolution. Since exception handling is typically poorly tested (FU et al., 2005, SINHA and HARROLD, 2000), it is likely that the faults associated to undetected exceptional conditions remained dormant in the source code and were only discovered when field failures occurred.

Furthermore, faults of omission impose a difficult challenge to developers, since they are caused by the absence of elements in the source code. This way, developers are required to be aware of which exception handling dependencies they were supposed to implement and to detect their absence in the source code. For some exception handling dependencies, current exception handling mechanisms already provide support for developers. For example, the reliability checks performed at compile-time by Java raises the awareness of developers for the need of proper handlers for checked exceptions. Even so, developers still introduce faults related to the lack of handlers, such as those related to the faults classified as "Missing handler". For the other exception handling dependencies, there is no support to warn developers about the absence of expected dependencies in the source code. The existence of re-mappers and raisers in the source code, for example, is not verified by current exception handling mechanisms. And faults classified as "Missing raiser" and "Missing re-mapper" were frequent in this study. Therefore, it seems interesting to provide support for checking the existence of expected raisers and re-mappers in the source code.

3.3.2 Difficulty in Detecting Exception Handling Faults

Once exception handling faults are better known, automated solutions for detecting them in the source code can be provided. In programming languages that perform exception handling using return codes (e.g., C), the faults related to exception handling can be automatically detected by statically analyzing the source code and searching for specific structural patterns (BRUTNIK et al., 2006). Our initial aim with this study was to better understand what categories of exception handling faults occur in programs following the *try-catch-throw* model. This way, we expected to lay the foundations for further investigations on the detection and repair of exception handling faults.

However, in this study, it was not possible to generalize specific structural patterns in the source code related to the categories of exception handling faults. On the contrary, we observed ambiguities in some structural patterns. We observed that a specific structural pattern in the source code was related to the cause of one fault, whereas for another fault the same pattern was used to repair it. For instance, generic *catch* blocks and empty *catch* blocks were related to the causes of “Incorrect handler” faults, but they were also used to repair “Missing handler” faults. Thus, simply searching for generic or empty *catch* blocks could pinpoint to many false positives, including *catch* blocks that were introduced in the source code to repair previous faults.

Detecting exception handling faults based on structural patterns would be even worse for faults of omission. For example, “Missing re-mapper” faults are caused by the absence of re-mappers. To detect potential faults of this category, every *catch* block handling an exception can be considered as a potential place where a re-mapper was not implemented. Similarly, every method propagating an exception can also be considered as a potential place where “Missing re-mapper” faults might be occurring. Given the abundance of global exceptions being propagated through multiple methods in a system, this strategy would yield many false positives. This would also happen for the other faults of omission, such as “Missing raiser” and “Missing handler”. Therefore, the detection of exception handling faults in programs following the *try-catch-throw* model based on the search for structural patterns would yield many false positives, or would be limited to very specific faults.

The detection of faults of commission requires detecting where in the source code exception handling dependencies are implemented incorrectly. Similarly, detecting faults of omission requires detecting where exception handling dependencies are missing in the source code. However, without prior knowledge about the intended exception handling design of a system, one cannot be sure whether an existing exception handling dependency is correct or not by only reviewing the source code. Also, without knowing the intended exception handling design, one cannot know whether exception handling dependencies are missing in the source code. In other words, one can only know whether exception handling dependencies are correct or whether they are missing in the source code if the exception handling design of a system is known. For this reason, it is important to explicitly define and document the intended exception handling design of a system.

In fact, we observed developers describing in commit messages and bug reports that some faults occurred due to violations of exception handling policies unknown to other developers. An exception handling policy refers to the set of design decisions governing how exception handling should be implemented. These policies were not documented in the projects, but were known by core developers, who helped to clarify them to other developers. Some of these comments were:

> *(Container) should throw the correct exception if an application attempts to modify the associated JNDI context*
>
> *execute() cannot throw JasperException, so it matches the signature for Task.execute()*

In the previous comments, developers explained that faults were caused because exceptions were raised with incorrect types. We also observed faults due to exceptions caught in the wrong place, exceptions not re-mapped as expected, and the like. These faults occurred because developers violated policies defining which exception types should be raised, the places where exceptions should be handled, how exceptions should be re-mapped, respectively. Thus, detecting violations of exception handling policies may be an alternative to detecting exception handling faults based on the search of structural patterns in the source code.

3.3.3 All Faults Related to Global Exceptions

In this study, we observed that all exception handling faults analyzed were related to global exceptions. This may point to the fact that the majority of exceptions in the target systems were global, or that dealing with global exceptions is more error-prone than dealing with local exceptions. In any case, this points to the importance of taking into consideration the impact of global exceptions.

In fact, implementing exception handling in the presence of global exceptions is not an easy task for most developers, since it requires being aware of several global design decisions of the software system. These decisions comprise where exceptions should be handled, which exception types should be used to raise exceptions, where and how exceptions should be re-mapped, among others. And all these decisions are important parts in the implementation of exception handling dependencies for global exceptions. In large software systems, developers typically work in the source code of specific modules. Thus, they are not always aware of the global design decisions governing global exception in a system. For this reason, it is important to define and document the design decisions governing how exception handling for global exceptions should be implemented. It is also important to check whether these decisions are obeyed in the source code. As observed in the faults analyzed in this study, failing to realize these decisions correctly, or completely missing them, may end up introducing faults in the source code.

3.3.4 Harmful Exception Handling Negligence

Currently, software systems are commonly implemented following incremental development processes: software systems are gradually developed in incremental versions, in which each increment adds to the previous version new features and refinements to existing features. As a consequence, new exceptions may emerge along the software evolution, requiring that the related exception handling dependencies be identified and implemented *a posteriori*. By only reviewing the source code and bug reports, it is difficult to analyze what motivated developers during software construction. That is, it is difficult to judge whether the faults analyzed in this study were due to negligence of developers in the early versions of the target systems, or because the appropriated exception handling dependencies were only well-understood or discovered later during software evolution. Even so, there were few cases that seemed to point to cases of actual negligence.

A symptom of exception handling negligence could be observed in the co-occurrence of faults classified as “Incorrect re-mapping” and faults classified as “Missing handler”. In particular, 3 faults were classified as “Incorrect re-mapping” because exceptions were re-mapped to unchecked exception types. In addition, these faults were also related to the “Missing handler” category because the re-mapped unchecked exceptions were left uncaught. This scenario suggests that developers were trying to avoid the automatic reliability verification performed by Java, since unchecked exceptions are not verified at compile-time. Developers seem not to recognize that this practice may harm the robustness of their systems. Without explicit policies prohibiting or discouraging these practices, developers are free to implement them. Thus, developers introduce these faults in the source code without being aware of their threats to robustness until failures occur.

3.4 Threats to Validity

This section discusses the study limitations based on the threats to the study validity, presenting the measures took to mitigate these threats.

3.4.1 Construct Validity

Threats to the construct validity relate to the identification of the exception handling faults. The process of identifying exception handling faults relied on a search heuristic that matches specific commit messages patterns and the keyword "exception". Only the keyword "exception" was used because we wanted to observe only the commits that developers considered important to explicitly mention that the modification performed was related to an exception. We are aware that we could have used other words, such as "fault" and "failure", for example. We considered that if the developer did not use the word "exception" in the comment, then he might not have considered that commit as related to exception handling. However, we are aware that we may have missed commits also related to exception handling, but whose commit messages did not contain the keyword "exception". With these strict selection criteria we aimed at augmenting the confidence in observing commits actually related to exception handling faults.

Furthermore, the data generated by these heuristics was manually reviewed to discard false positives. Commits that matched the searched patterns were manually inspected to discard those that were not related to faults. In particular, we only considered for analysis commits related to bug reports explicitly marked by the development teams of the analyzed systems as being related to faults. Thus, we trusted in opinions that were not biased towards the goal of our study. In addition, faults were considered as being related to exception handling if the observed changes performed in the source code involved the modification, removal or addition of any exception handling dependency. Due to the manual inspection of each commit identified, its respective bug report and source code modifications, there is a certain degree of confidence that the analyzed faults were actually related to exception handling faults.

3.4.2 Internal Validity

One threat to the internal validity of the study stems from possible biases in the categorization of the analyzed exception handling faults. The analyzed faults were classified according to the exception handling dependency implemented and the fault type observed (faults of commission or faults of omission). During the analysis method, the researcher manually inspected the source code of both the faulty and repaired revisions of the target systems. Based on this observation, the researcher classified a given fault according to objective criteria, as defined in Section 3.1.2. Also, the commit messages and the bug reports were also inspected to support the classification performed by the researcher. And it is worth noticing that the developers who produced these artifacts were not aware of the goals of this study. This way, biases from the viewpoint of the researcher were mitigated by using objective classification criteria and other sources of unbiased information.

Another threat to the internal validity of the study relates to the number of exception handling faults analyzed. As we previously discussed, we adopted a heuristic strategy with strict criteria for identifying exception handling faults. The adopted heuristic may have missed commits that were related to exception handling faults, but whose messages did not match the patterns searched. It is worth noticing that it was not possible to first identify bug reports related to exception handling and then identify which modifications in the source were performed for this bug report. This was not possible because most bug reports are not synchronized with the version control system of the projects. That is, bug reports do not have links to the repairing modifications in the source code. Thus, scanning through commit messages searching for links to bug reports was the option left. We are aware that the strict criteria adopted may have reduced the size of the sample analyzed. We adopted strict criteria to achieve higher confidence in the results, since there was little previous knowledge about exception handling faults. We favored the confidence in the results achieved, at the cost of possibly missing other exception handling faults. Therefore, the results achieved by this study are consistent, although possibly not complete, since other exception handling faults in the target systems were possibly not analyzed.

3.4.3 External Validity

This study focused on open-source projects implemented in Java. Therefore, the study findings are still hardly generalizable to other kinds of software projects and other programming languages, especially those that provide exception handling mechanisms with different characteristics. In addition, this study adopted procedures for data analysis based on manual inspection of the data. By manually inspecting the collected data we aimed at improving the confidence in the results achieved, at the price of hindering the scalability of the study to larger samples. In other words, the internal validity of the study was favored in detriment to its external validity.

In order to promote the generalizability of the categories of exception handling faults identified in this study, the faults were classified based on generic fault types and exception handling dependencies that are commonly implemented in programming languages following the *try-catch-throw* model. It is worth highlighting that 53 mainstream programming languages follow the *try-catch-throw* model. So this exception handling model is representative of mechanisms most frequently used by developers nowadays to achieve software robustness. It is expected that the proposed categories are generic enough to cover exception handling faults identified in different programming languages following the *try-catch-throw* model. In fact, an independent study conducted by other researchers also identified categories of exception handling faults and their categories can be subsumed by the categories presented in this study. Even categories of exception handling faults identified in programs that do not follow the *try-catch-throw* model can be subsumed by the categories identified in this study. This is further discussed in Section 3.5.

3.5 Related Work

As far as we are concerned, our study was the first to conduct a thorough analysis to categorize exception handling faults in programs following the *try-catch-throw* model. An existing study was conducted in the context of C programs using return-code idioms to implement exception handling (BRUTNIK et al., 2006). Although the analysis of our study was conducted in the context of programs following the *try-catch-throw* model, it is still possible to trace some links between the fault categories identified in our study and those observed in programs handling exceptions using the return-code idiom. Brutnik et al. identified a list of fault categories in this idiom-based exception handling strategy. The fault categories identified by the authors were: “Function does not return”, “Wrong error variable returned”, “Assigned and logged value mismatch”, “Not linked to previous value” and “Unsafe assignment”. The “Function does not return” category, for example, is defined as follows:

“(this fault category) occurs when a function declares and uses an error variable (i.e., assigns a value to it), but does not return its value”.

In other words, faults in this category occur due to the lack of a return statement indicating an error code. This category is equivalent to our “Missing raiser” category. Similarly, the “Wrong error variable returned” category is defined as follows:

“(this fault category) occurs when a function declares and uses an error variable but returns another variable”.

That is, faults in this category occur due to a return statement indicating the incorrect error code. This fault category is equivalent to our "Incorrect raiser" category. Moreover, their "Assigned and logged value mismatch" and "Not linked to previous value" categories are equivalent to our "Incorrect handler" category and their "Unsafe assignment" category is equivalent to our "Incorrect raiser" category. This evidences that the proposed classification is able to cover exception handling faults even in programs implementing exception handling with return-code idioms.

Current studies conducted in the context of programs following the *try-catch-throw* model focused only on specific categories of exception handling faults. Coelho et al. assessed exception handling faults in the context of AspectJ programs (COELHO et al., 2008). The authors observed faults related to uncaught exceptions being introduced because aspects did not capture exceptions as intended. These aspects were not capturing exceptions in the intended places due to the improper definition of their point-cuts. They also observed that uncaught exceptions were created because aspects explicitly raised them, but no proper *catch* blocks existed in the source code. According to our classification, the faults identified by Coelho et al. can be categorized as "Missing handler". Thus, the analysis conducted in their study was restricted to only one of the fault categories identified in our study.

In our empirical studies conducted in collaboration with Cacho et al., we assessed exception handling faults in the context of Java and C# programs (CACHO et al., 2014a, CACHO et al., 2014b). The exception handling faults observed in this study were related to uncaught exceptions originated from re-mappings. These faults can be categorized as "Missing handler", and to "Incorrect re-mapper". There were also faults related to handling actions raising exceptions that were left uncaught. We did not observe faults similar to these in this study. This fault category could be categorized as "Missing handler", due the uncaught exceptions, and also as "Incorrect handling action", in case the handling actions raising the exceptions were incorrectly implemented. Thus, the analysis conducted in these studies was also restricted to three fault categories identified in this study.

It would be interesting to revisit these previous studies by considering all categories of exception handling faults identified in this study. These studies concluded that the number of exception handling faults increased as systems evolved and that developers introduced faults in the systems as they modified exception handling code along software evolution. These conclusions were made by considering only a small set of fault categories. By considering a wider range of fault categories, these results could be even more alarming, since other faults

Table 3.2: Comparison with the work of Ebert et al.

Our Classification	**Classification proposed by Ebert et al.**
Incorrect handler	Empty catch block
	Error in the handler
	Exception caught at the wrong level
	General catch block
	Catch block where only a finally would be appropriate
Incorrect raiser	Error in the exception assertion
	Exception that should not have been thrown
	Wrong exception thrown
Incorrect re-mapper	Wrong encapsulation of exception cause
Incorrect terminating action	Error in the cleanup action
Missing handler	Lack of a handler that should exist
Missing raiser	Exception not thrown
Missing terminating action	Lack of a finally block that should exist
Missing re-mapper	-
-	Error in the definition of exception class
-	Inconsistency between source code and API documentation

may have occurred in the analyzed systems but were not taken into account during the data analysis.

After our study, Ebert et al. (EBERT et al., 2015) conducted a similar investigation about exception handling faults. The authors performed a survey with developers and analyzed bugs reported in Tomcat and Eclipse. By combining the exception handling faults pinpointed by participants of the survey and the faults analyzed in the bugs, the authors proposed a set of 15 categories of exception handling faults. Table 3.2 compares the exception handling faults classification proposed in our work to the classification proposed by Ebert et al.. As one can observe in this table, their classification is similar to ours. From the classification proposed by Ebert et al., five categories are equivalent to the sub-categories of the “Incorrect handler” category, three are equivalent to the sub-categories of the “Incorrect raiser” category. Moreover, one category identified in our study were not present in theirs: “Missing re-mapper”. Also, two categories proposed in their study were not observed in ours: “Error in the definition of exception class" and "Inconsistency between source code and API documentation". According to our classification criteria, these categories would be called as "Incorrect exception type definition" and "Incorrect exception documentation".

3.6 Summary

Better understanding exception handling faults is important to avoid their introduction and to detect them in the source code. This chapter presented an empirical study conducted in the context of two open-source projects to further investigate what categories of exception handling faults occur. We collected and analyzed bug reports and commits related to exception handling faults to gather empirical knowledge about them. The exception handling faults analyzed were classified according to the exception handling dependency implemented. They were also analyzed to check if they were related to the absence of exception handling dependencies or to the existence of incorrect dependencies (Section 3.1.2). This analysis resulted in a set of 9 different categories of exception handling faults (Section 3.2.2). This categorization is the first contribution of this thesis.

Initially, our goal with this study was to better understand what categories of exception handling faults occur in programs following the *try-catch-throw* model, so that we could support their detection and repair. However, we observed in this study that most exception handling faults were not related to structural patterns in the source code, such as empty or generic *catch* blocks (Section 3.3.2). Therefore, supporting their detection by only analyzing the source code structure, as performed in programs that implement exception handling using return-based idioms (BRUTNIK et al., 2006), would yield many false positives or would be restricted to very specific cases.

We also observed that exception handling faults occurred because developers violated implicit exception handling policies (Section 3.3.2). These policies were not documented in the projects, but were known by core developers, who helped to clarify them to other developers. In particular, we observed faults due to exceptions caught in the wrong place, exceptions not re-mapped as expected, exceptions raised with incorrect types, and the like. And all these faults occurred because developers were unaware and violated policies defining the places where exceptions should be handled, how exceptions should be re-mapped and which exception types should be raised, respectively. These results motivated us to move our goal towards the investigation of means to support detecting and repairing violations of exception handling policies.

Chapter 4 presents the proposed solution to support the explicit definition of exception handling policies and the detection of violations in the source code. Chapter 5 presents the proposed solution for assisting the repair of these violations. Finally, it should be noted that the results presented in this chapter were published in a paper (BARBOSA et al., 2014).

4 Specifying and Verifying Exception Handling Policies

Although exception handling is central to robust software development, most software projects still deal with exceptions without explicit exception handling policies (DELEMOS and ROMANOVSKY, 2001, KIENZLE, 2008). In this thesis, we define an *exception handling policy* as the set of design decisions governing how exception handling should be implemented in a system. Developers participating in recent surveys reported that there exist exception handling policies in their systems, although not much effort is spent in documenting them (EBERT and CASTOR, 2013, EBERT et al., 2015). Most of the times these policies are not documented and exist as implicit rules in the source code (BUSE and WEIMER, 2008, THUMMALAPENTA and XIE, 2009). In fact, in the study presented in Chapter 3, we observed that some faults occurred due to violations of implicit exception handling policies of the projects. These policies were not documented and only core developers seemed to be aware of them.

The lack of explicit exception handling policies may have negative consequences in software systems. Unaware of the design decisions governing how exception handling should be implemented, developers implement it following *ad-hoc* strategies. Following a policy-ignorant strategy will likely induce developers to simplify the exception handling implementation in their programs (Section 2.2.1), probably introducing violations in the source code. In fact, without knowing the exception handling policy of their systems, developers cannot even realize that they are introducing violations in the source code. Since exceptions are expected to occur rarely during program execution, the exception handling code is rarely exercised. And as exception handling is poorly tested in software systems, exception handling violations remain dormant in the source code (FU et al., 2005, SINHA and HARROLD, 2000). For this reason, these violations are only discovered later when they cause field failures.

Even if software designers and developers are keen on explicitly defining exception handling policies for their systems, there is still no proper support for that (Section 2.3.2). They can try to use exception handling mechanisms in programming languages, but these mechanisms are not intended to express design decisions of how exception handling should be implemented. At best, software designers and developers can express their exception handling policies in unstructured documents in natural language, such as comments in the source code. But these forms are not really useful in supporting the detection of exception handling violations. For this reason, software projects should have explicit exception handling policies. More important than that, these policies should support the detection of violations in the source code.

In this context, this chapter presents the proposed solution that addresses the first research question of this thesis:

> ***RQ1.*** *How to support the definition and checking of exception handling policies in the source code?*

In order to address the research question RQ1, we proposed the **E**xception handling **P**olicies **L**anguage (EPL), a specification and verification language for exception handling policies in software projects. EPL supports the detection of exception handling violations in the source code, which is the first part of the research goal of this thesis. Exception handling violations are detected by specifying exception handling policies and statically analyzing the source code to check its policy adherence. Thus, developers can detect exception handling violations in the source code early in the development process, avoiding that they remain dormant in the source code and cause failures.

The design of EPL was inspired by deontic logic and the fault types presented in the previous chapter (faults of commission and faults of omission, as defined in Section 3.1.1). Deontic logic is a branch of modal logic which focuses on the study of norms expressed in terms of the concepts of *obligation* and *permission* (CHELLAS, 1980). We borrow these concepts to express exception handling policies as obligations and permissions over the structural dependencies established between software modules and exceptions.

In addition, we designed EPL so that violations of obligations and permissions are aligned with the definitions of faults of omission and faults of commission, respectively. In EPL, a violation to an obligation means that a given module was obligated to establish an exception handling dependency with an exception, but it did not. For example, a module was obligated to re-map from an API-specific exception to an application-specific exception, but it did not implement this re-map in the source code. Similarly, a violation to a permission means that a given module is establishing an exception handling dependency with an exception, but it is not allowed to. For example, a module is only allowed to re-map from `SQLException` to `PersistenceException`, but it actually implemented a re-map from `SQLException` to `RuntimeException`, which it is not allowed to. Therefore, using obligations and permissions to define an exception handling policy and enforcing policy adherence in the source code, developers can detect violations that may lead to failures in their systems.

In addition, we designed and implemented EPL as a domain-specific language. Domain-specific languages are programming or specification languages that offer expressive power focused on a particular problem domain (VANDEURSEN et al., 2000, FOWLER, 2010). These languages provide a bare minimum of features that allow solutions to be expressed with vocabulary of terms and level of abstraction compatible with the problem domain. We expressed obligations and permissions using the modal verbs "Must" and "May", respectively. We also used a vocabulary of terms that are commonly employed in exception handling mechanisms in programming languages, such as "Propagate", "Handle", "Raise", etc. Thus, we aimed at designing EPL so that policy specifications produced were easy to write and read to developers minimally acquainted with exception handling mechanisms in programming languages.

EPL was evaluated with a user-centric study and a case study. In the user-centric study, we evaluated the "definition of exception handling policies" part of the research question RQ1. This study was conducted with developers and consisted of an observational study followed by semi-structured interviews. With the observations and experiences gathered in this study, we could better understand the trade-offs related to different language design decisions. Thus, we identified some language characteristics that hindered the definition of policies and that motivated new language constructs. In the case study, we evaluated the "checking exception handling policies in the source code" part of the research question RQ1. We performed a case study with one open-source project and two industry-strength systems to investigate if and how violations detected by EPL relate to exception handling faults. The results showed that violations detected by EPL and faults in exception handling share common causes. Therefore, exception handling violations can be used to detect potential causes of exception-related failures.

The rest of this chapter is structured as follows. Section 4.1 how exception handling policies are specified in EPL and Section 4.2 presents how these policies are verified. Then, Section 4.3 presents a user-centric evaluation of EPL and Section 4.4 presents the case study conducted to evaluate EPL. Finally, Section 4.5 presents related works and Section 4.6 summarizes this chapter.

4.1 Making Exception Handling Policies Explicit

As previously discussed, EPL was designed as a domain-specific language for exception handling policies. In particular, policies in EPL are expressed as constraints over the exception handling dependencies that modules establish with exceptions. In programs implemented with exception handling mechanisms following the *try-catch-throw* model, exception handling dependencies are established at the method level. In other words, methods are the source code elements that raise, handle, propagate, re-throw or re-map exceptions. However, defining a system's exception handling policy at the method level would not scale well, since systems have a large number of methods. In order to support the specification of exception handling policies at a higher level of abstraction, EPL provides the *Compartment* construct. This construct is further discussed in Section 4.1.1.

In EPL, constraints are expressed with the *Rule* construct. These constraints are expressed in terms of permissions, prohibitions obligations about how compartments and specific exception types can establish exception handling dependencies. The rule construct is further discussed in Section 4.1.2. Finally, in Section 4.1.3 we present the *Alias* construct, which allows the definition of an alias for a list of exceptions in the specification of exception handling policies.

4.1.1 Compartments

In EPL, a compartment is a named and referable entity that comprises a set of methods in the source code. Compartments may be specified in two ways: by explicitly listing the names of their elements or by defining type constraints for their elements. When a compartment is defined by explicitly listing its elements, it is defined with the following syntactic structure:

```
define <elements> as compartment <comp_id>;
```

In the previous syntactic structure, <`elements`> specify a list of elements in the source code that compose the compartment and <`comp_id`> specifies an identifier for the compartment being defined. To define a list of elements in the source code, the wildcard character "*" may be used to define a name pattern. The following example depicts how a compartment may be defined at a fine-grained level:

```
define pucrio.DataAccess.create*,
        pucrio.DataAccess.read*,
        pucrio.DataAccess.update*,
        pucrio.DataAccess.delete*
        as compartment DATA-ACCESS;
```

In the previous example, the *DATA-ACCESS* compartment comprises all methods within the module `pucrio.DataAccessor` whose fully qualified names have the prefix `create`, `read`, `update` or `delete`.

Moreover, compartments can also be defined in terms of more coarse-grained elements, as shown in the next example:

```
define pucrio.controller.*.*
        as compartment CONTROLLER;
```

In the previous example, the *CONTROLLER* compartment comprises all methods of all modules whose fully qualified names has the prefix `pucrio.controller`.

Compartments in EPL may also be specified by defining type constraints for their elements. In particular, EPL supports the definition of compartments in terms of subtype relations, as shown in the following example:

```
define X.* as compartment CONTROLLER
        where X is subtype of IController;
```

In the previous example, the *CONTROLLER* compartment comprises all methods within modules that are subtypes of the `IController` type. EPL also supports the definition of compartments by combining name patterns and subtype relations, as shown in the next example:

```
define X.create*, X.read*, X.update*, X.delete*
        as compartment DATA-ACCESS where X is
        subtype of IDataAccess;
```

In the previous example, the *DATA-ACCESS* compartment comprises all methods within modules that are subtypes of the `IDataAccess` type and whose fully qualified names have the prefix `create`, `read`, `update` or `delete`. It is worth mentioning that this feature for defining compartments in terms of subtype relations was not in the first version of EPL; we identified the need for this feature during one of our case studies (Section 4.4).

4.1.2 Rules

The main purpose of an exception handling policy is to explicitly define constraints over the exception handling dependencies that source code elements establish with specific exception types. These constraints are expressed in

EPL in terms of permissions and obligations. More specifically, the rule concept expresses permissions and obligations about how compartments can establish exception handling dependencies with specific exception types. A given compartment C establishes an exception handling dependency with a specific exception type E if there exists one of C's element that establishes an exception handling dependency with an exception of the type E. Moreover, exception handling dependencies can be established between code elements and exception types when a code element handles, raises, propagates, re-maps or re-throws an exception of a given type. These are "canonical" structural dependencies between exceptions and code elements, since they are typical dependencies with which developers structure their exception handling code (Chapter 2). Table 4.1 summarizes these exception handling dependencies.

Table 4.1: Exception handling dependencies supported by EPL

Exception Handling Dependency	**Description**
m handles E	Method m handles an exception of type E in its scope
m raises E	Method m explicitly raises an exception of type E in its scope
m propagates E	Method m propagates an exception of type E from its scope
m re-maps from $E1$ to $E2$	Method m re-maps an exception of type $E1$ to an exception of type $E2$ in its scope
m re-throws E	Method m re-throws an exception of type E from its scope

EPL provides three rule types to express permissions: *Only-May*, *May-Only* and *Cannot* rule types. Rules of the *Only-May* type express permissions about which compartments can establish exception handling dependencies with specific exception types. This rule type is expressed with the following syntactic structure:

```
only <comp_id> may <dep_type> <exception_list>
```

The *Only-May* rule type is defined in terms of an exception handling dependency (<`dep_type`>), which may be one of the dependencies shown in Table 4.1 and a list of exception types (<`exception_list`>). For example, the *Only-May* rule supports the definition of the following permission between a compartment X and exception types A, B and C:

```
only X may handle A, B, C;
```

In the previous example, the compartment named X is the only one in the specification that has permission to handle exceptions of type A, B and C. If a compartment different from X handles A, B or C, then this is considered a violation of the specified rule. The semantics of the *Only-May* rule type is the same for the other exception handling dependencies.

Another rule type provided by EPL to express permissions is the *May-Only* rule type. Rules of this type express permissions about which exception types a given compartment can establish exception handling dependencies. This rule type is expressed with the following syntactic structure:

```
<comp_id> may only <dep_type> <exception_list>
```

The *May-Only* rule type is syntactically similar to the previous rule type. It is also defined in terms of an exception handling dependency and a list of exception types. The *May-Only* rule supports the definition of the following permission between a compartment X and exception types A, B, C and D:

```
X may only remap from A to B, from C to D;
```

The compartment named X in the previous example has permission to re-map only from exceptions of type A to exceptions of type B and from exceptions of type C to exceptions of type D. If the compartment X performs a re-mapping that is neither from type A to type B nor from type C to type D, then this is considered a violation of the specified rule. The semantics of the *May-Only* rule is the same for the other exception handling dependencies.

Notice in the previous example that the *Re-map* exception handling has a syntactic structure slightly different from structure used in the example of the *Only-May* rule type. The argument <`exception_list`> for the *Re-map* exception handling dependency is defined in terms of pairs (E_1, E_2). Each pair specifies the exception type being caught (E_1) and the exception type that the caught exception is supposed to be re-mapped (E_2). Each pair is expressed in EPL with the syntactic structure: `from` E_1 `to` E_2. This syntactic structure for the *Re-map* exception handling dependency is the same for the other rule types.

EPL also provides a rule type to express permissions in a negative form. That is, it allows expressing prohibitions. The *Cannot* rule type is used to express rules that prohibits compartments to establish exception handling dependencies with specific exception types. The *Cannot* rule was not part of the first version of EPL; we identified the need for this type of rule during our user-centric study (Section 4.3). The *Cannot* rule type is specified with the following syntactic structure:

```
<comp_id> cannot <dep_type> <exception_list>
```

The syntactic structure of the *Cannot* rule type is similar to the structure of the other rules types. The *Cannot* rule type can be used to express prohibitions as follows:

```
X cannot raise A, B, C ;
```

In the previous example, the compartment named X is prohibited to raise exceptions of type A, B and C. If compartment X raises exceptions of type A, B or C, then this is considered a violation of the specified rule. The semantics of the *Cannot* rule is the same for the other exception handling dependencies.

Besides allowing expressing permissions, EPL also provides a rule type to express obligations. The *Must* rule type allows expressing obligations that a given compartment has to establish with specific exception types. The *Must* rules are defined with the following syntactic structure:

```
<comp_id> must <dep_type> <exception_list>
```

The *Must* rule type has syntactic structure similar to the other rule types: it has an exception handling dependency and a list of exception types as argument. The *Must* rule can be used to express obligations as follows:

```
X must propagate A, B, C;
```

The compartment named X in the previous example is obligated to propagate exceptions of type A, B and C. If compartment X does not propagate exceptions of type A, B and C, then this is considered a violation of the specified rule. The semantics of the *Must* rule is the same for the other exception handling dependencies.

4.1.3 Alias for Exceptions

EPL also provides a language construct to allow users to define an alias for a given list of exception types. In fact, this language construct was also not part of the first version of EPL; its need also emerged during our user-centric study (Section 4.3). We defined the alias construct with a syntactic structure similar to the structure used to define compartments. Aliases for lists of exceptions are defined with the following syntactic structure:

```
define <exception_list> as alias <alias_id>;
```

This new language construct allows simpler specifications, as shown in the following example:

```
define IOException, RecordStoreException as alias API-
   EXCEPTIONS;
```

```
DATA-ACCESS cannot raise API-EXCEPTIONS;
DATA-ACCESS cannot handle API-EXCEPTIONS;
only CONTROLLER may handle API-EXCEPTIONS;
```

In the previous example, the alias *API-EXCEPTIONS* groups the exception types `IOException` and `RecordStoreException`. Then, the same alias is used to specify different rules, avoiding the repetition of the same list of exceptions in the rules definition.

4.2 Verifying Exception Handling Policies

To verify a given exception handling policy, we implemented the *EPL Verifier*, which verifies exception handling policies for Java programs. The *EPL Verifier* consists of two main modules: the *Rule Checker* and the *Facts Extractor*. The *Rule Checker* receives the policy specification and for each specified rule it uses the *Facts Extractor* to check whether there exists methods in the source code violating the rules. For each violated rule, the *Rule Checker* module presents a list of the methods in the source code violating the rule. Next, Section 4.2.1 details the *Facts Extractor* module and Section 4.2.2 details how the *Rule Checker* works.

4.2.1 Extracting dependency facts

The *Facts Extractor* was implemented using the Eclipse Java Development Tools (JDT). It analyzes the source code of a system to extract the information needed by the *Rule Checker* module. In particular, the source code of a system is parsed and its abstract syntax tree is analyzed in order to extract dependency facts related to the exception handling dependencies supported by EPL. To extract the dependency facts related to the exception handling dependencies supported by EPL, the *Facts Extractor* analyzes the *catch* blocks, *throw* statements and *throws* clauses in the source code.

In the context of the *Facts Extractor* module, *catch* blocks may be related to the *Handle*, *Re-map* and *Re-throw* exception handling dependencies. The *Facts Extractor* module registers only one dependency fact for each *catch* block. The following pseudocode shows how it distinguishes each case:

```
IF catch-block contains throw-statement THEN
   IF throw-statement raises the same exception instance caught
       by the catch block THEN
      Register Re-throw fact
   ELSE
      Register Re-map fact
   END IF
ELSE
   Register Handle fact
END IF
```

As defined in the previous pseudocode, if a given *catch* block contains a *throw* statement, then the *Facts Extractor* module registers either a *Re-throw* or a *Re-map* fact; otherwise, the *Facts Extractor* module registers a *Handle* fact. If the *throw* statement contained by the *catch* block raises the same exception captured by the *catch* block, then the module registers a *Re-throw* fact; otherwise, the module registers a *Re-map* fact. The rationale behind this characterization of the facts associated to *catch* blocks is that an exception is only handled when the program execution flow returns to its normal flow. When the caught exception is re-mapped or re-thrown, the program continues in its exceptional flow. Therefore, the caught exception is not actually handled.

In the context of the *Facts Extractor*, *throw* statements may also be related to more than one exception handling dependency; they may be related to the *Raise*, *Re-map* and *Re-throw* dependencies. The following pseudocode depicts how the *Facts Extractor* distinguishes each case:

```
IF throw-statement is inside catch block THEN
   IF throw-statement uses the same exception instance caught
      by the catch block THEN
      Register Re-throw fact
   ELSE
      Register Re-map fact
   END IF
ELSE
   Register Raise fact
END IF
```

If a given *throw* statement occurs inside a *catch* block, then the *Facts Extractor* module registers either a *Re-throw* or a *Re-map* fact; otherwise, the module registers a *Raise* fact. If the *throw* statement inside a *catch* block raises the same exception instance caught by the *catch* block, then the module registers a *Re-throw* fact; otherwise, the module registers a *Re-map* fact.

Finally, from the *throws* clauses, the *Facts Extractor* module extracts the facts related to which exceptions are explicitly propagated by a given method. The following code snippet exemplifies the dependency facts extracted by the *Facts Extractor* module:

```
public void foo() throws IOException, MyException2,
   MyException3{
   try{ throw new FileNotFoundException(); }
   catch( MyException1 e ){ /*handle exception*/ }
```

```
    catch( MyException2 e ){ throw e; }
    catch( MyException3 e ){ throw new MyException3(); }
}
```

In the previous example, the *Facts Extractor* module analyzes the *throws* clause to extract the facts related to the *Propagate* exception handling dependency. Thus, the module registers that the `foo()` method establishes *Propagate* dependencies with the `IOException`, `MyException2` and `MyException3` types. It should be noted that, in this Java variant of EPL, *Propagate* rules are intended to specify how exceptional interfaces of methods should be declared. Rules of this type are not intended to specify which specific exceptions flow through the boundaries of a given method. In the previous example, the type `IOException` is declared in the method exceptional interface, but the exception that is actually raised and flows through the boundaries of the method is `FileNotFoundException`. To specify which exceptions *throw* statements should raise, one should use *Raise* rules. For this reason, the dependency facts related to the *Propagate* dependency are extracted directly from the *throws* clause; the *Facts Extractor* module does not employ flow analysis techniques to extract more accurate information about the exact types of the exceptions flowing through the boundaries of methods.

The *Propagate* rules are intended to specify how exceptional interfaces should be declared because these interfaces are an important part of the exception handling code of Java programs. In fact, a significant part of the maintenance effort related to exception handling in Java programs is spent on maintaining the exceptional interface of methods (BARBOSA and GARCIA, 2011, CACHO et al., 2014a). If a developer declares a given exception type on his method's exceptional interface and needs to change this interface during software evolution, then this will result in changes to different parts of the code. Therefore, deciding which exception types are allowed to be declared on the exceptional interface of a method is an important design decision. For this reason, it needs to be specified and verified throughout the software development process.

In addition, EPL does not support the specification of rules defining which specific exceptions flow through the boundaries of a method because this requires knowing the internal structure of modules, which would break the information hiding principle. Thus, by focusing on the specification and verification of rules related to the *Propagate* dependency type only in terms of the *throws* clause of Java methods, we allow developers to make internal choices in their methods, as long as they adhere to the external behavior specified in the exception handling policy. Also, there are innumerous possible unchecked exceptions flowing through the boundaries of methods, including `NullPointerException`, `BufferOverflowException`, `Arithmetic-Exception`, and many others. Therefore, specifying all possible exceptions flowing through the boundaries of methods would be impractical. In fact, this is one of the reasons of why unchecked exceptions are not required to be declared in exceptional interfaces in Java (JSL-6), as discussed in Chapter 2.

Still on the previous code snippet, the *Facts Extractor* module registers for the first *catch* block the fact that the `foo()` method establishes a *Handle* dependency with the `MyException1` type. For the second *catch* block, which contains the second *throw* statement, the module registers the fact that the `foo()` method establishes a *Re-throw* dependency with the `MyException2` type. For the third *catch* block, which contains the third *throw* statement, the module registers the fact that the `foo()` method establishes a *Re-map* dependency, in which it re-maps an exception of the `MyException3` type to another exception of the same type. It is worth highlighting that the *Re-map* facts registered by the *Facts Extractor* module are defined in terms of the exception type declared in the *catch* block and the exception type of the *throw* statement.

To extract the dependency facts related to the *Raise* and *Re-map* dependencies, the *Facts Extractor* module analyzes the type of the *throw* statement expression. In the previous example, the type of the *throw* statement expression can be statically determined by only inspecting the *throw* statement: its expression is a new instance creation, so its type is the type of the instance being created. In the previous example, the only exception type that can be raised by the first *throw* statement is the `FileNotFoundException` type. Therefore, the *Facts Extractor* module registers the fact that the `foo()` method establishes a *Raise* dependency with the `FileNotFoundException` type. However, when the expression of *throw* statements refer to method calls or conditional expressions, the type of the raised exception cannot be directly determined by only inspecting the *throw* statement. For this reason, a type-inference algorithm is necessary to determine the type of the raised exception in these cases. The following code snippet exemplifies these cases:

```
public void bar() throws Exception {
   if( condition1 ){
      Exception e1 = new MyException();
      throw e1;
   }
   else{
      Exception e2 = cond() ? new Type1Exception() :
         createException();
      throw e2;
   }
}
public Type2Exception createException(){
   return new Type2Exception();
}
```

In the previous example, the expressions of both *throw* statements are references to variables. To determine the exception type actually raised by the *throw* statements, we implemented the type-inference algorithm for exception types proposed by Sinha and Harrold (SINHA and HARROLD, 2000). The algorithm performs a reverse data-flow analysis starting from the *throw* statement, searching for statements that assign a type to that variable. For the first *throw* statement in the previous example, the algorithm finds an assignment expression for the `e1` variable, whose right-hand side is a new instance creation expression. Thus, the type of the raised exception can be precisely determined and the *Facts Extractor* module registers the fact that the `bar()` method raises `MyException`, which is the type of the instance being created.

For the second *throw* statement, the type-inference algorithm also finds an assignment expression for the `e2` variable, but the right-hand side of the variable assignment is a conditional expression. Since the assignment of the `e2` variable depends of a conditional expression, its type cannot be precisely defined statically. Consequently, the type of the exception raised by the second *throw* statement cannot be precisely defined either. In these cases, the type-inference algorithm returns the set of the possible types of the *throw* statement expression. Thus, in the previous example, the type-inference algorithm finds two possible assignable types for the `e2` variable: `Type1Exception`, which comes directly of the *then* expression of the conditional expression; and `Type2Exception`, which comes from the *else* expression of the conditional expression – the returned type of the `createException()` method invocation. Then, for each possible type of the *throw* statement expression, the *Facts Extractor* module registers a fact. Thus, it registers the fact that the `bar()` method establishes *Raise* dependencies with both `Type1Exception` and

`Type2Exception`. Similarly, if the type-inference algorithm finds more than one possible type for a *throw* statement that is part of a re-map, then the *Facts Extractor* will register more than one *Re-map* fact, one for each possible type of the *throw* statement.

Finally, it is worth mentioning that the type-inference algorithm implemented by the *Facts Extractor* module simplifies its analysis when a virtual method invocation is found in its data-flow path. In Java, every non-static method is by default a virtual method, except final and private methods (JSL-6). Moreover, in object-oriented paradigm, a virtual method is a method whose behavior can be overridden within an inheriting class by a method with the same signature to provide polymorphic behavior. Therefore, given a virtual method invocation, it is not alway possible to statically decide which concrete method is being invoked. To overcome this limitation, when the *Facts Extractor* module finds a virtual method invocation in the data-flow analysis path of a *throw* statement, it does not try to analyze all return statements of all possible virtual method invocations to determine the precise type being returned. Instead, it considers the return type in the method's signature. This simplification may return less precise types, but it is at least type-safe, since the precise types are either the actual type or subtypes of the type considered by the *Facts Extractor* module. In fact, this simplification is similar to the analysis performed by the Java compiler. Moreover, empirical evidence suggests that the overwhelming majority of *throw* statement expressions in Java programs are new instance expressions (SINHA and HARROLD, 2000), so in most cases the types of raised exceptions can be precisely determined without loss of precision.

4.2.2 Checking the rules

The *Rule Checker* module checks for each specified rule in the exception handling policy whether there exist in the source code violating facts. Then, for each violated rule, the verifier presents a list of the violating facts. In EPL, each rule type specifies how a given compartment is allowed to establish an exception handling dependency to a list of exception types. Consider a rule R that specifies how a compartment C is allowed to establish a dependency D to a list E of exception types. A violation of a *Cannot* rule is defined as follows:

$\exists m$:`Method` $\in C \wedge \exists\ e$:`Exception` $\in E$ | $D(\ m,\ e\ )$

In the previous notation, $D(m,e)$ means that a method m establishes an exception handling dependency D with the exception type e. Thus, a violation of a rule R of the *Cannot* type occurs when there exists a method m in compartment C that establishes an exception handling dependency D to an exception type e in the list E. In other words, a violation of a *Cannot* rule occurs when a method establishes an exception handling dependency to an exception type that it is prohibited to.

Similarly, violations of *May-Only* rules are defined as follows:

$\exists\ m$:`Method` $\in C \land \exists\ e$:`Exception` $\notin E \mid D(\ m,\ e\ )$

A violation of a rule R of the *May-Only* type occurs when there exists a method m in compartment C establishing an exception handling dependency D with an exception type e not in the list E. That is, the method establishes an exception handling dependency to an exception type that it is not allowed to.

Violations of *Only-May* rules are defined as follows:

$\exists\ m$:`Method` $\notin C \land \exists\ e$:`Exception` $\in E \mid D(\ m,\ e\ )$

A violation of a rule R of the *Only-May* type occurs when there exists a method m not in compartment C establishing an exception handling dependency D with an exception e that is in the list E. That is, an *Only-May* rule R states that only methods in C are allowed to establish an exception handling dependency D with the exception types in E, but a method not in C is establishing an exception handling D with an exception type in E.

Finally, violations of *Must* rules are defined as follows:

$\nexists\ m$:`Method` $\in C \land \exists\ e$:`Exception` $\in E \mid D(\ m,\ e\ )$

A violation of a rule R of the *Must* type occurs when for at least one exception type e in the list E there is no method m in compartment C establishing an exception handling dependency D with e. In other words, for at least one exception type e specified in the list E, there is no method fulfilling its obligation of establishing an exception handling dependency D with e.

Verification warnings

EPL makes it possible to define sets of inconsistent rules. Developers might specify rules that conflict with one another. Prior to checking for policy violations, the *Rule Checker* module will validate the given set of specified rules and warn the developer of any conflicts between rules. Developers must then correct these conflicts before checking their policies. Thus, readers of the specification can readily comprehend the intended use of exceptions without the extra burden of understanding the complete specification and identifying implicit conflicts. The following rule conflicts are detected by the *EPL Verifier*.

Conflict between Cannot and Must, May-Only, Only-May. Given the same compartment and the same exception handling dependency, a *Cannot* rule and a *Must* rule will conflict if they share the same exception type. Likewise, a *Cannot* rule will conflict with *May-Only* rules and *Only-May* rules when referring to a common compartment and exception type. For example, the following cannot rule:

```
X cannot handle A;
```

Conflicts with:

```
X must handle A;
X may only handle A;
only X may handle A;
```

Conflict between Only-May and Only-May. Given a common exception handling dependency and two or more different compartments, *Only-May* rules will conflict if they refer to a common exception type. For example, the following *Only-May* rules conflict with each other:

```
only X may raise A;
only Y may raise A;
```

Conflict between Must and May-Only. Given a common compartment, a *Must* rule and a *May-Only* rule will conflict if an exception is declared in the *Must* rule, but it is not declared in the *May-Only* rule. For example, the following rules conflict with each other:

```
X must handle A;
X may only handle B, C;
```

Redundancy between Cannot and Only-May. Besides the previous conflicts between rules, *Cannot* rules and *Only-May* rules may also interact with each other to create implicit redundancies in the specification. A given *Cannot* rule and a given *Only-May* rule will create an implicit redundancy if they refer to different compartments and to a common exception type. The following rules create a redundancy in a specification:

```
only X may handle A;
Y cannot handle A;
```

In the previous example, the rule *"Y cannot handle A"* is subsumed by the rule *"only X may handle A"*, i.e., the second rule does not add practical information to the policy specification. The *Rule Checker* will warn developers about these implicit redundancies in the specification. Unlike conflicts between rules, developers are allowed to verify their policies even if their specifications contain redundant rules. Since redundancies between *Only-May* and *Cannot* rules do not introduce inconsistencies in the policy specification, we allow developers to leave redundant rules in the specification as a manner to make them more explicit to other readers of the specification.

4.3 User-Centric Evaluation

We designed EPL as a domain-specific language in an attempt to promote its acceptance among developers. For this reason, we designed a study to investigate the acceptance of EPL. In addition, we were also interested in investigating to what extent the language actually provides suitable constructs for specifying exception handling policies. To investigate the design of EPL, we conducted a user-centric study that consisted of an observational study followed by interviews. In the next sections, we detail the settings of our observational study (Section 4.3.1), we present its results (Section 4.3.2) and discuss the threats to the study validity (Section 4.3.3).

4.3.1 Settings of the Study

This section describes the settings of the user-centric study. First, we present the goal of the study and the research questions addressed. Then, we detail the design of the study.

Goal and Questions

As previously discussed, EPL is a domain-specific language aimed at providing support to developers define exception handling policies. The support for defining exception handling policies is the first part of the research question RQ1. To actually support developers in this task, it is important that EPL provide constructs that are indeed able to express exception handling policies and that these constructs are well-accepted by developers. In this direction, we designed EPL as a domain-specific language for exception handling policies in an attempt to improve its acceptance among developers. In this context, the goal of this study is stated as follows:

> ***Goal:*** *Analyze EPL for the purpose of understanding its acceptance from the viewpoint of potential users of the language.*

This study aims at understanding the trade-offs related to decisions in the language design and how these decisions may have affected the acceptance of the language. Thus, we refined the following research question from the previous goal:

What are the factors that influence the acceptance of EPL?

With the previous research question, we aimed at investigating factors in the language design that would affect the acceptance of EPL and that could inspire improvements in the language.

Study Design

In this study, we recruited developers with different backgrounds and from different organizations and asked them to use EPL in an observational study, which was followed by a semi-structured interview. We opted to combine these two research methods for two reasons. First, we could set up scenarios in which participants could use EPL while we directly observed them. Second, we could use the interview to collect their experiences in using EPL.

Observational Study. The sessions of the observational study were performed individually. All participants had access to the same artifacts and used the same target system. Thus, it would be possible to compare the specifications produced by participants. Each session comprised two tasks of 30 minutes each. The goal of the tasks was to mimic different scenarios of use of the proposed language. This way, we could observe how participants used the language in the different scenarios.

The first task simulated a scenario where the exception handling policy is specified when the system is already in production, but without an explicit policy. In this case, it is necessary to recover the exception handling policy from the source code. The goal of participants was to inspect the source code of the target system and infer an exception handling policy from the source code. The source code of the target system was available as a project in the Eclipse IDE and participants were allowed to use any feature of the IDE. Participants were free to specify their policies as they wanted to, as long as they used only the source code. In fact, they had no access to any type of documentation of the target system. We believe that this would be the most common scenario of use of the language, since documentation artifacts about exception handling policies are currently not part of most software projects (DELEMOS and ROMANOVSKY, 2001, KIENZLE, 2008).

The second task simulated a scenario where the exception handling policy is specified during the design of the system, prior to its implementation. This scenario is less usual, but it happens in some software projects with more critical robustness requirements (BRITO et al., 2009). In this task, participants received the system documentation. The system documentation described the intended architecture of the target system, showing its main components in a component diagram, as well as the intended dependencies between these components. The documentation also described the intended exception hierarchy tree, showing the hierarchy structure in a class diagram. Finally, the documentation described the exception handling responsibilities of each component regarding each exception type. Participants were in charge of planning and producing the intended exception handling policy of the target system based solely on the documentation of the target system; they had no access to its source code.

Prior to actually performing the tasks, the researcher gave to each participant a lecture (approximately 20 minutes) about: basic exception handling concepts, the concepts provided by EPL, the study settings and the architecture of the target system. The presentation covered all main topics of EPL: compartments, types of rules and exception handling dependencies supported.

Participants had at their disposal in both tasks a notebook with a regular text editor to produce the specification. They also had a printed document containing the description of the core concepts of the language, the language grammar and a concrete example of an exception handling policy specification. The concrete example was defined to cover all concepts provided in the specification language. This way, participants were exposed to all main features of EPL: compartments, rules and dependency types. During the lecture and in the printed document provided, we intentionally did not mention the possibility of specifying conflicting rules. We did not mention this because we wanted to observe whether participants would be aware of possible conflicting rules. After the lecture and before performing the first task, participants were allowed to read the document describing the language without a limit of time. During the tasks, the researcher only observed the participants and did not participate in the task. Also, participants were not allowed to ask questions to the researcher during the tasks.

Interview. The goal of the interview was to collect participants experiences' and opinions in using the specification language. To help us in understanding participants' usage of the EPL language, we relied on a technology acceptance model that try to explain the determinant factors of technology usage behavior. We built our interview guide based on the Technology Acceptance *Model* (TAM) (DAVIS, 1989). The TAM is one of the most used models for predicting technology adoption, but in the context of this study this model was useful to set up the theoretical background of our interview guide. The TAM considers that two dimensions may influence the user behavior towards a technology: *"Perceived ease of use"* and *"Perceived usefulness"*. The first dimension relates to how much a user believes that using a given technology is free from effort. The second dimension relates to how much a user believes that using a given technology is useful to support his tasks. Moreover, we structured our interview guide as a semi-structured interview to have flexibility to explore unforeseen information that could emerge during the interviews.

Data Collection and Analysis Method

There were three main data sources in this study: field notes taken during the tasks performed by participants, the specifications produced by participants during the tasks and the answers to the post-task interview. During the observational study, the researcher took field notes about the specific operations that participants performed during each task. Examples of these notes are *"Participant highlighted the system documentation"*, *"Participant used the search feature of the IDE"*, etc. There were also notes with addiitonal questions to be asked during the post-task interview.

The specifications produced by participants were saved at the end of each task and later analyzed by the researchers. For each specification produced, we computed its size in terms of the number of compartments and rules specified. We also compared the specifications of each task in terms of how compartments were defined and in terms of what rule types and exception handling dependencies were used. We also assessed the specifications to check whether they were consistent to the source code and documentation of the target system. In particular, we checked if the exception types, compartments names and exception handling dependencies specified by participants were present or not in the source code and documentation of the target system.

The post-task interviews were recorded and later transcribed by the researcher. The interviews were performed in Portuguese. The audio transcriptions were also in Portuguese, but they were translated to English by the researcher in order to report the results.

To analyze the interview transcriptions, we adopted an iterative coding process. First, we extracted the main fragments of participants answers and assigned a topic to each of these fragments. A topic refers to a name created by the researchers for a common and recurring theme that clusters a set of fragments. Next, we reexamined the assigned topics and further examined the fragments and field notes to check whether new topics emerged. We also checked the need to merge existing topics in more abstract topics. We repeated these last steps until we reached saturation, i.e., until new topics did not emerge and existing topics could not be merged in more abstract topics.

Target System

To produce the artifacts required to perform the study, we needed a software system of which we had access to the source code and its intended exception handling policy. We used the Mobile Media system as the target system because it is a well documented system that has been used in previous empirical studies that assessed its software architecture (ARCOVERDE et al., 2013, MACIA et al., 2012, MACIA et al., 2012a) and its exception handling implementation (CACHO et al., 2008, COELHO et al., 2008, SALES and COELHO, 2011). Thus, there was sufficient information about Mobile Media to infer its exception handling policy. The artifacts describing Mobile Media's exception hierarchy tree and responsibilities of components regarding exception types did not exist prior to this study. The researcher produced them by examining the source code and existing artifacts of Mobile Media.

Participants

The participants of our study were invited by email and voluntarily accepted to participate on the study. We invited participants from different organizations and with different levels of experience. Participants in our study had their university education in different institutions in different cities. Their experience ranged from very inexperienced, with less than one year of experience in industry, to very experienced, with more than ten years of experience in industry. They also had different backgrounds in terms of their previous experience in performing relevant design decisions in software projects.

To keep participants' anonymity, we refer to each one of them with an *Id*, as shown in Table 4.2. The table also presents the participants profiles in terms of their years of development experience. It also describes the programming languages that they use (or have already used) in their project activities. All participants had previous experience with the use of exception handling mechanisms in their projects and they all have different backgrounds in terms of experience with programming languages. All participants worked with Java, but also worked with other programming languages. Each participant had already used at least two programming languages implementing exception handling mechanisms with different characteristics. For instance, in Java, the compiler automatically verifies whether there exist proper handlers for

Table 4.2: Participants Profile

Id	**Experience in Software Industry (in years)**	**Programming languages**
P1	<1	Java, C, C++, C#
P2	10	Java, JavaScript, C#, PHP
P3	8	Java, JavaScript, PHP
P4	3	Java, C#
P5	3	Java, JavaScript, PHP
P6	10	Java, JavaScript
P7	8	Java, C++, C#
P8	6	Java, JavaScript, C#, VB.NET
P9	5	Java, JavaScript, C#, Delphi
P10	7	Java, C#, VB.NET

checked exceptions. In C#, JavaScript and PHP, on the other hand, there exist exceptions, but no automatic verification for proper handlers.

4.3.2 Data Analysis and Results

This section presents the analysis of the collected data and the results of our user-centric study. First, we present the analysis of the specifications produced in each task of the observational study. Second, we present the analysis of our observations on how developers used the language in each task. Finally, we present the analysis of the interviews.

Artifacts Analysis

In this section, we present the analysis of the specifications produced by participants during the observational study. Next, we detail the analyses of the artifacts produced in each task of the observational study.

Specifications Produced During the First Task. Comparing the specifications produced during the first task, we could observe that participants produced policy specifications with similar compartments, but with very different rules. As can be observed in Table 4.3, the number of specified compartments varied in a smaller range (minimum of 2, maximum of 6) than the number of specified rules (minimum of 2, maximum of 30).

The source code used in the study comprised 5 high-level packages: *Controller*, *Screen*, *AlbumData*, *ImageAccessor* and *ImageUtil*. We observed that all participants defined compartments for the *Controller* and *AlbumData* packages. The packages *ImageAccessor* and *ImageUtil* were specified as compartments by 5 participants; the other participants did not specify compartments

Table 4.3: Specifications Produced in the First Task

	# Compartments	**# Rules**
Min	2	2
Max	6	30
Median	4	5

for these packages. Similarly, the *Screen* package was specified as a compartment by 4 participants; the other participants did not specify compartments for this package. Only participant P2 specified a compartment named *Main* for the main class of the system.

In addition, most participants defined their compartments at the package level with name patterns using the wildcard operator. Only participant P1 defined his compartments by listing all of its elements. It is worth mentioning that the feature for specifying compartments in terms of subtype relations was incorporated to EPL only after this observational study. For this reason, participants did not use it in this study.

For the specified rules, one can observe in Table 4.3 that the maximum number of rules specified in a specification was 30, which was produced by participant P2; the other participants produced specifications with 2 to 9 rules. While participant P2 was performing the first task, we observed that for the rules that could be specified using a list of exceptions, he specified one rule for each exception in the list. The following code snippet exemplifies what participant P2 did. Participant P2 specified a set of rules similar to the following rules:

```
X must raise A;
X must raise B;
X must raise C;
```

Instead of specifying a single rule similar to the following rule:

```
X must raise A, B, C;
```

If participant P2 had defined rules using exception lists, his specification would have 9 rules, instead of 30. It is worth noticing that what participant P2 did is not an error in the language use, but rather an ineffective use. We asked him why he adopted this approach and he answered:

> *P2: I thought that each rule had to have only one exception. Could I have used a list here? (...) Well, now I can see here in the examples (provided with the language documentation) that there is an example with a comma and a list. That (using a list) would decrease the number of lines (of the specification), because it is all repeated in here.*

In terms of the exception handling dependencies used, some policy specifications produced in the first task were defined in terms of only one dependency: participants P3 and P5 specified only rules of the *Handle* dependency, whereas participant P7 specified only rules of the *Raise* dependency. The specification produced by the participant P2 was the only one to cover all the exception handling dependencies provided by EPL. The specifications produced by the other participants comprised no more than two different exception handling dependencies, mostly *Handle* and *Raise*.

Similarly, in terms of the rule types used, some specifications were defined in terms of only one rule type: participants P5 and P8 produced their specifications using only rules of the *May-Only* type, whereas participants P6 and P9 produced their specifications using only rules of the *Must* type. There was no specification produced using only the *Only-May* type. The other participants used more than one rule type to produce their specifications. It is worth mentioning that the *Cannot* rule type was incorporated to EPL only after this observational study, so participants did not use it in this study.

We also reviewed the specifications produced in the first task to check if they were consistent with the exception handling code implemented in the target system. That is, if the specified rules referred to source code elements, exception handling dependencies and exception types that existed in the source code. We observed that all rules specified by participants P2, P3, P4 and P7 were consistent with the source code. For the other participants, there were both consistent and inconsistent rules in their specifications. There was no participant that produced only inconsistent rules.

It is worth highlighting that inconsistent rules are not necessarily incorrect in this context. During this study, participants were free to produce their specifications as they wanted to. We did not give any specific instruction of how participants should produce their specifications. This means that some participants may have produced an exception handling policy that directly mirrors the information contained in the source code. That is, the specification produced strictly describes what is implemented in the source code. On the other hand, other participants may have considered that the source code does not necessarily adhere to an exception handling policy. That is, they may have produced an idealized exception handling policy that should have been followed in the target system implementation, instead of strictly mirroring the source code. For this reason, we did not consider the consistency between the specification produced and the source code as an indicator of the correctness of the specification. It only points to different approaches adopted by participants in this task.

Table 4.4: Specifications Produced in the Second Task

	# Compartments	**# Rules**
Min	0	6
Max	4	21
Median	4	7

Specifications Produced During the Second Task. For the specifications produced in the second task, we can observe in Table 4.4 that there was one specification without any compartment definition, which was produced by participant P9. The other participants produced specifications with the number of compartments definitions ranging from a minimum of 3 and a maximum of 4. While participant P9 was performing the second task, we observed that he specified his rules without specifying any compartment. During the interview we asked him why he did not specify compartments and participant P9 answered:

P9: Well, I think I forgot to specify them. Now that you've asked me, I realized that I defined the rules using the names of the components (in the diagram of the documentation).

We can also observe in Table 4.4 that the maximum number of rules specified in the second task is 21. This specification was produced by participant P2, who instead of specifying his rules using lists of exceptions, created one rule for each exception in the list, as we previously discussed. The other participants produced specifications with the number of rules ranging from a minimum of 6 rules to a maximum of 9 rules.

In the system documentation provided in the second task, there were 5 exception handling requirements that could be specified by participants. In general, participants specified more than 5 rules because they expressed the same requirement more than once using different rule types. Consider the following requirement that was present in the system documentation provided to participants:

The following exceptions are raised by third party APIs and are handled in the context of the Image Acessor component: RecordStoreException, RecordStoreNotOpenException, IOException.

The previous requirement was specified as two rules by participants P2 and P4:

```
only IMAGE_ACCESSOR may handle RecordStoreException,
   RecordStoreNotOpenException, IOException;
IMAGE_ACCESSOR must handle RecordStoreException,
   RecordStoreNotOpenException, IOException;
```

As one can observe in the previous specification, participants P2 and P4 expressed the previous requirement using rules *Only-May* and *Must* rules for the same requirement. Both rules were consistent with the system documentation.

Participants P2, P3, P4 and P6 covered all exception handling requirements. They specified each requirement by using rules that were consistent with the system documentation. Participants P1, P5, P7, P9 and P10 also specified rules that covered all exception handling requirements, but each participant specified one rule incorrectly. Curiously, they all mistook the specification of a rule related to the same requirement; they forgot to specify some exceptions in a given rule. This requirement was specified in two different parts of the system documentation and participants seemed to miss the second part of the requirement. Participant P8 completely missed one requirement, but specified rules that were consistent with the documentation of the other requirements. Moreover, only participant P5 created rules that were not related to any of the exception handling requirements defined in the system documentation. This participant specified two extra rules, which were both inconsistent with the system documentation. In particular, participant P5 created two rules for exceptions that were not defined in the system documentation and these exceptions did not seem to represent cases of other existing exceptions misspelled.

By comparing the specifications produced in the second task, we observed that they were similar to each other. Except for the participant P9, who forgot to specify the compartments, all the other participants grouped the system components into compartments in the same way: they specified 4 compartments comprising the same components. In fact, we observed that participants P4 and P8 initially specified 4 compartments, but they removed the specification of one of their compartments at the end of the task. We asked them why they removed it and they answered that, at the end of the task, they realized that they had not produced any rule for that compartment, so they opted to remove its definition. Moreover, all participants used at least three dependency types (*Handle*, *Raise*, *Re-map*) to produce their specifications.

We could also observe that when the system documentation explicitly used a modal verb to define a given exception handling requirement, all participants produced the same rule type. For example, the system documentation had one requirement explicitly using the modal verb *must*:

> *The exceptions raised by third party APIs do not leave the Image Accessor component and must be remapped to the PersistenceMechanismException type.*

For this requirement, all participants specified it as a *Must* rule. However, when requirements in the system documentation did not explicitly use a modal verb, participants specified the same requirement with different rule types. For instance, the system documentation had one requirement stating the following:

> *The Image Accessor component raises the following exceptions: NullAlbumDataException, ImageNotFoundException (...).*

Participants P1, P3, P5 and P10 specified the previous requirement with a *May-Only-Raise* rule, whereas participants P2, P4, P8 and P9 specified it using a *Only-May-Raise* rule and participants P6 and P7 specified it using a *Must-Raise* rule.

Finally, except for the lapse of participant P9 in the second task, we did not observe any other serious mistakes in the specifications produced in both tasks. The most recurring mistake observed was the lack of the *"from"* keyword in the rules of the *Re-map* dependency type. A total of 6 out of the 10 participants forgot this keyword in an at least one rule in their specifications. In addition, the other mistakes we found were minor syntax errors, such as a missing semicolon or a misspelled keyword.

Observation Analysis

By observing how participants used the language during the observational study, distinct approaches in each task could be observed. Next, we detail the approaches observed in each task of the study.

Approaches Adopted During the First Task. During the first task, when participants had to inspect the source code to infer the exception handling policy, the approaches adopted by participants to produce the policy specification varied. We could observe that some participants adopted systematic approaches to inspect the source code, whereas others seemed to inspect the source code at random. We also observed that those who adopted systematic approaches relied on search features of the IDE to assist them.

Participant P2 navigated through the packages of the system and found the exception types defined by the application. Then, for each of these exception types, he used the *"References in Project"* search feature of the Eclipse IDE, which shows the places in the source code where a given type is used. Participants P3 and P8 searched in the source code methods matching the keyword *"catch"*. Similarly, participant P7 searched in the source code methods matching the *"throw new"* keywords. Then, for each matching method, participant P7 used the *"Call Hierarchy Tree"* view of the Eclipse IDE, which shows the callers and callees of a given method.

Participant P6 followed a systematic approach during the first task, but he did not use any search feature of the IDE. First, he defined one package of the system as one compartment. Next, he opened each one of the classes of this package and inspected its source code. When he finished inspecting the classes of the first package, he repeated these steps for another package. For the other participants, we could not observe any structured approach to inspect the source code and produce the exception handling policy specification. They also did not use any specific feature of the IDE, rather than those generally used to navigate through the source code files.

The different approaches adopted in the first task may be one of the reasons of why the specified rules produced in this task were so different, as discussed in the previous section. Participants P3 and P7, for instance, produced specifications with rules of only one exception handling dependency, the *Handle* and *Raise* dependencies, respectively. This is actually aligned with their systematic approach of searching for keywords related to specific exception handling dependencies. Participant P2, on the other hand, was the only participant to produce a specification covering all exception handling dependencies provided by EPL. This is also aligned with his approach of searching for all the references of a given exception type in the source code, instead of searching for a specific dependency. In addition, participants P2, P3 and P7 produced policy specifications that were completely consistent with the source code. So it might be the case that their systematic approaches were employed to produce specifications that mirrored the information contained in the source code.

Approaches Adopted During the Second Task. During the second task, when participants had to produce the exception handling policy based on the system documentation, we could observe that most participants followed a similar approach. First, they specified the compartments of the system based on the components diagram provided in the system documentation. Then, they inspected the system documentation searching for requirements that could be expressed as exception handling rules. For each requirement found, they specified one or more rules. The only exception to this approach was the participant P9, who forgot to specify his compartments, as discussed in the previous section. He specified his rules using the names of the components shown in the system documentation, instead of the names of compartments.

The similarity in the approaches adopted by most developers in the second task may be the reason why the specifications produced were so similar. In particular, all participants defined their compartments matching the architecture elements described in the system documentation. In terms of how participants specified their rules, except for the case in which the modal verb *must* was explicit in the text, as discussed in the previous section, we could not distinct how each developer interpreted the system documentation to produce their rules.

Interview Analysis

The analysis of the interview transcripts supported the answer to the question of this study:

What are the factors that influence the acceptance of EPL?

In this study, we extracted 133 fragments from the interviews transcriptions and a total of 6 topics emerged from these fragments. These topics point to factors that seemed to influence the acceptance of EPL. The topics that emerged, presented in order of largest number of associated fragments, were: *Perceived Usefulness* (29 fragments), *Expressiveness* (26 fragments), *Usability* (25 fragments), *Impact on Performance and Productivity* (22 fragments), *Learnability* (16 fragments) and *Comprehensibility* (15 fragments). Next, we present quotes that illustrate the participants' reactions about different dimensions of EPL captured by the six topics.

Perceived Usefulness. The *Perceived Usefulness* topic groups together the fragments in which participants mention whether and in which ways the EPL would be useful to them. Overall, all participants considered the proposed language useful, but in different ways. Participant P10, in particular, commented about the usefulness of expressing exception handling policies:

> *P10: I think that defining this (exception handling) policy is like defining any other system requirement. If we don't specify it, products will have to conform to what? If this (exception handling policy) is part of system's specification, then we have to do it. Conformance (to requirements) is part of the system's quality.*

Participants P1, P2, P7, P8 and P10 mentioned that using the language would prevent the introduction and assist the detection of problems related to exception handling in the source code:

P1: It would help me to avoid bad programming practices and follow the rules someone else specified, which would probably enhance my knowledge about how exceptions have to be used. In fact, using this language would help me or guide me on how exceptions must be used, how things should be done.

P2: It would improve source code quality, because we wouldn't make many mistakes (related to exceptions), because there would be a warning in my IDE "hey, look, you raised an exception in the wrong place!" Then, in an inspection meeting there would be probably less problems (related to exceptions) to find in the code.

P7: This is another way of checking documentation versus source code. To check if what was specified is really implemented in terms of exception handling. Specially in systems with complex exception handling it is important to have this type of mechanism to avoid problems.

P7: (It would be useful) Specially in languages without checked exceptions. With checked exceptions you kind of know where exceptions are, when you don't (have checked exceptions) anything can happen. So when you don't have policies for exceptions, when they are used randomly, it's really easy to make things wrong. And it's really hard to find this later.

P10: When you define a policy you can check if code is correct. We can even detect faults, like detecting that one exceptions is there and nobody is catching it, you know? Besides defining policies, it is also a way for detecting some faults in the code.

These comments made by participants P1, P2, P7, P8 and P10 are aligned with the aim of EPL, which is defining and enforcing exception handling policies as a means to detect exception handling violations in the source code. Moreover, participant P3 introduced another perspective about how the language can be useful. This participant considered that using the language is useful because it would raise the awareness of a role required in the development team in charge of managing exception handling in the software project:

P3: I believe that using the language would be useful because we would have a person to explicitly think about exception handling

> *policies. This is perhaps the main benefit: a person to define and reason about the exception handling policy and the exception handling rules.*

Participant P6 also considered the language useful, but he showed concerns about aligning this type of solution with the organizational objective:

> *P6: The language seems useful and I would like to see its practical use. The problem is that in most cases there is no support from the company administration to invest on this (type of solution). They only focus on results. Apparently there is a barrier in using these "auxiliary methods" because it seems a waste of time, a waste of money. But I do like things aimed at improving software quality.*

Finally, participants mentioned other reasons why they considered the language useful, including: it may help improving system's architecture by defining exception handling responsibilities, it may support communication about exception handling among members of a development team, it may support system comprehension and it may support maintenance of legacy systems.

Expressiveness. The *Expressiveness* topic groups together the fragments in which participants commented on whether it was possible to express what they wanted using the EPL language. Overall, all participants considered that expressing exception handling policies with the proposed specification language was easy. However, participants also mentioned issues with the language that hindered its expressive power. In particular, six out of ten participants mentioned the lack of a proper construct in the language to express negation. About this matter, participant P1 said:

> *P1: During the second task I read in the documentation (of the target system) "this compartment (Screen) does not raise any exception". But then I looked the language grammar and I couldn't find anything for that. There was no "not" to express "may not handle" or "may not raise".*

Participants P2 and P4 mentioned that they could not express the rule *"Screen does not raise any exception"*, but did not explicitly complain about the lack of a specific construct for that. Instead, they tried to express it using the *Only-May* rule. Participant P2 said:

P2: If I specify that only compartment X may raise a given exception, then Screen is not allowed to raise that exception. That's how I tried to express this rule, using the only-may rule.

Even though the understanding of participant P2 about the *Only-May* rule type is correct, his attempt to express the rule *"Screen does not raise any exception"* using the *Only-May* rule had some unwanted side effects in the specification. Participant P2 tried to express the previous rule as follows:

```
only X may raise *;
```

In the previous rule, the wildcard operator was used to specify that the compartment *X* is the only one allowed to raise any exception. As a consequence, the *Screen* compartment is not allowed to raise any exception. This specifies the intended exception handling rule for the *Screen* compartment, but it may have some unwanted side effects, since no other compartment is allowed to raise any exception. In fact, participant P2 produced another rule of the *Raise* dependency that conflicted with this one, although he seemed not to realize that. For this reason, users may misuse the *Only-May* rule in an attempt to compensate the lack of a negation construct in EPL, at the risk of producing conflicting rules, which shows a design flaw of the proposed specification language. When we first defined EPL, the negation construct was actually not defined. We thought that compartments were supposed to be specified in terms what they should do, instead of in terms what they should not do. It was only during the post-task interviews that we realized that the negation construct was necessary in EPL. Hence, the negation construct is an improvement that was incorporated *a posteriori* into EPL.

Usability. The topic *Usability* groups together the fragments in which participants mentioned issues related to the practical use of EPL. Some of the aspects about how participants used EPL were already described when we discussed the artifact and observation analyses. During the interviews, all participants considered EPL easy to use. However, some participants pondered that reasoning about the exception handling policy was not easy, specially during the first task. About this, participants P4 and P5 said:

P4: I think that it would be difficult for a user to retrieve a set of (exception handling) rules from the source code of a previously implemented system. But I can't think of an easy way of doing it. I don't think that the language itself can ease that.

> *P5: Specifying it (the exception handling policy) was difficult, specially because I didn't know the system. But if you know what you have to specify, using the language for that is very simple.*

Participant P3 mentioned that the complexity involved in extracting exception handling rules from the source code relates to the amount of information involved:

> *P3: (It is difficult, but) Not because of the language, but because of the huge amount of information (in the source code) that I had to deal with.*

Currently, EPL still does not support extracting exception handling rules from the source code to support developers while they produce their policies. There exist automated techniques that mine exception handling rules from the source code (THUMMALAPENTA and XIE, 2009). Also, exception handling policies are often aligned with architectural design rules, so techniques for recovering architectural design rules from the source code (GARCIA et al., 2013) can also assist developers in extracting exception handling policies from the source code. These solutions could be used to extract exception handling rules that express how exception handling is implemented in the source code. These rules would assist developers in reasoning how the source code is currently implemented and how it should be implemented. Then, they would be able to produce their policies without wasting much effort in inspecting the source code.

Finally, participants identified some usability aspects of the proposed specification language that could be improved. Participant P1, in particular, suggested the possibility of defining an alias for a list of exceptions:

> *P1: It would be great if we could define a name for a given list of exceptions the same way we define a compartment. Without this, I waste my time re-writing all exceptions over and over. It would decrease the copy and paste and would be faster to specify.*

Participant P3 also complained about having to re-write the same list of exception names in different rules along the specification, even though she did not suggest a possible solution. An alias construct for lists of exceptions was not part of the first version of the language. We considered it a good suggestion and added it to EPL, as previously described. Finally, it is worth noticing that during the interviews no participant commented on the possibility of specifying inconsistent rules, which reinforces the need for warning users about their occurrence.

Impact on Performance and Productivity. The topic *Impact on Performance and Productivity* groups together the fragments in which participants mention the impact that using EPL would have on their activities. Overall, participants considered that the language would have a positive impact on their performance and productivity. Most of the reasons given by participants were already discussed in the topic *Perceived Usefulness.* Only participants P1 and P3 showed some concerns about a possible negative impact on their performance and productivity. About this matter, participant P3 said:

> *P3: For the person responsible for specifying the rules, it would be very costly. Not because of the language, which is simple to use, but as far as I know from the projects I've worked on, this concept of "exception policy" is simply not defined. In most cases we only handle exceptions locally, so it would be very hard to reason about broader policies.*

About the possible negative impact on performance and productivity, participant P1 said:

> *P1: Using the language may decrease a bit our productivity, but that's because developers have bad (programming) habits. So it would force us to adhere to the specified rules right from the start. In this manner, if you are a programmer with bad (programming) habits, then in the beginning it would take longer to get things done.*

Participant P1 explained that developers with bad programming habits would probably take longer to finish their implementation tasks. Later on, during the interview, participant P1 pondered his previous comment:

> *P1: Maybe I would take longer to get things done, but that's because I have these bad (programming) habits. In the long term, using the language would actually help me to adopt better (programming) habits, then probably my productivity would be the same or better.*

Learnability. The topic *Learnability* groups together the fragments in which participants mentioned their experiences to learn EPL. Regarding the learning of the specification language for exception handling policies, all but one participant of the study considered it easy to learn. Participants that considered the language easy to learn highlighted the conciseness of the language as the main factor of why it was easy to learn. The following quote from participant P2 summarizes this notion:

> *P2: The grammar is very small, so we don't have that high learning curve, such as when we learn a language like Java, which has a million different things to learn.*

Only participant P10 did not consider the language easy to learn. He considered that the time available was not enough to learn the language. He also complained about the lack of more examples illustrating the use of the language:

> *P10: With more detailed examples and with more time I think I would learn it better.*

Other than the issues related to the limited time and the lack of examples mentioned by participant P10, we did not observe any language characteristic that seemed to hinder its learning.

Comprehensibility. The topic *Comprehensibility* groups together the fragments in which participants commented on whether and how well they understood the elements of the EPL language. Regarding the comprehension of the concepts provided by the language, participants considered them easy to understand. Participants highlighted the importance of using common exception handling terms in the specification language. Participant P3 mentioned:

> *P3: Once you previously know basic exception handling concepts, you can easily understand them (concepts of the proposed language). (...) it was not difficult to memorize the "keywords" of the concepts, even in this short time of the task.*

There was only one participant that had difficulties in understanding the rule types. Participant P1, in particular, had difficulties in understanding the differences between the semantics of the rule types *Only-May* and *May-Only*. When asked why he faced difficulties in understanding these rule types, participant P1 said that the use of similar keywords for different rules confused him:

> *P1: If you had used another name, then I would have known right from the start "this one is this, that one is for that"; I would have mentally separated them.*

We intentionally designed these rules with similar keywords for the sake of uniformity in the language design. That is, these rules express permissions, so we used similar keywords. Apparently, the uniformity in the design of the rules created some confusion for participant P1. For this reason, when we added the new rule type with the semantics of prohibition, which is the negation of a permission, we opted to not use the keyword “May”. We tried to not overload the keyword “May” and, consequently, hinder the comprehension of the rule types. For this reason, we opted to express prohibition using the keyword “Cannot” instead of “May-Not”.

4.3.3 Threats to Validity

This section discusses the study limitations based on the threats to the study validity, presenting the measures took to mitigate these threats.

Construct Validity

Threats to construct validity relate to the way we investigated the acceptance of EPL. We are aware that innumerous factors influence users’ decisions about why, how and when they will use specific technologies, most of which were probably not covered in our study. We limited this threat by structuring the interview guide based on the *Technology Acceptance Model* (TAM) (DAVIS, 1989), which is an empirically tested model to study factors that influence technology adoption. This way, we tried to cover in our interview guide important factors that were tested in previous empirical studies. Moreover, we organized semi-structured interviews, so that we could ask follow-up questions and also questions that emerged during the observational study. Thus, we had the flexibility to ask questions whenever participants were sharing interesting information about topics not covered in our interview guide.

Internal Validity

A threat to the internal validity of the study relates to the target system used in the observational study. Instead of asking developers to produce exception handling policies for systems they already know, we asked them to produce policies for a system that they did not know before. We did this to avoid interference of the previous knowledge of each participant about its own system. It would be possible that experienced participants know their systems better than inexperienced participants that were working for less time in their systems. Thus, using a common and unknown for all participants mitigated this possible bias in the study, allowing that participants started the tasks in similar positions. Another possible threat to the target system relates to its medium size. However, we believe that this is not a major threat to our study. First, participants faced difficulties while inspecting the source code to extract the exception handling rules. For this reason, using a larger system would probably force developers to spend more time inspecting the source code. Thus, it would hinder the goal of the tasks, which was to expose developers to the use of the proposed language.

Another threat to internal validity is the possible cultural-related bias of participants. To mitigate this possible threat, we selected participants from different organizations and with different levels of experience. Participants also had their university education in different institutions. The specific organization culture or specific education training was not a significant threat to our study validity.

External Validity

Our research method does not support the generalization of the results to a general population of developers. Even so, with our user-centric study we could observe how participants used EPL and also characteristics of the language that seem to play important roles in developers usage. We could gather interesting insights that inspired concrete improvements in the language, as well as initial insights of possible approaches adopted by developers to produce exception handling policies. We could also understand the trade-offs related to different language design decisions based on concrete and well-documented experiences reported by participants.

4.4 Case Study

After analyzing EPL from a user-centric perspective, we designed a study to analyze it from another practical perspective. In particular, we designed a study to investigate whether and how violations of exception handling policies are related to exception handling faults. Next, Section 4.4.1 details the settings of the study, Section 4.4.2 presents the analysis of the collected data and Section 4.4.3 discusses the results and implications of this study.

4.4.1 Settings of the Study

This section describes the settings of the study conducted to analyze EPL from the user perspective. First, we present the goal of the study and the questions addressed. Then, we detail the design of the study.

Goal and Questions

As previously discussed, EPL is a domain-specific language aimed at supporting the checking of exception handling policies in the source code. The support for checking exception handling policies is the second part of the research question RQ1. The aim of checking exception handling policies in the source code is detecting exception handling violations that might induce system failures. In this context, the goal of this study is stated as follows:

> ***Goal:*** *Analyze EPL for the purpose of investigating the relations between exception handling violations and exception handling faults in the context of the source code of software systems.*

In the study presented in Chapter 3, we observed that exception handling violations caused faults in the analyzed systems. This motivated us to design EPL so that violations of obligations and permissions pointed to potential faults of omission and faults of commission, respectively. This study aims at further investigating whether violations detected by EPL and exception handling faults are indeed related. Thus, we refine the following questions from the previous goal:

> *Do violations detected by EPL and exception handling faults co-occur in the same methods?*

> *Are the causes of violations detected by EPL related to the causes of exception handling faults?*

These questions aimed at investigating whether and how violations detected by EPL are related to exception handling faults. In particular, to investigate wheter they co-occured in the same places in the source code and if their causes were related. This way, it is investigated wheter the violations detected by EPL are indeed able to point to potential causes of failures.

Study Design

To answer the questions of this study, we employed EPL to specify and verify exception handling policies of target systems and checked whether policy violations and exception handling faults co-occurred in the same methods. Then, we manually inspected the methods where violations and faults co-occurred to check whether the causes of the violations were related to the causes of the faults.

Next, the target systems selected for this study are presented. Then, the procedures conducted to collect and analyze the data are described.

Target Systems

To collect the data required for this study, it was necessary violations of exception handling policies and exception handling faults. To collect data related to violations of exception handling policies, first, it was necessary to have systems with known exception handling policies. However, as previously discussed, many software projects still do not explicitly define their exception handling policies. To mitigate this limitation, we used as target systems in this study well-documented projects, so that we could infer their exception handling policies from this documentation.

The first target system of this study was the Apache Tomcat system. We used Tomcat as the target system for two main reasons. First, Tomcat is endorsed by Oracle as the reference implementation of the Java Servlet and Java Server Pages technologies. These technologies are part of the Java Enterprise Edition – JEE – architecture, which has a complete and public available specification. Thus, we could extract exception handling-related requirements from the JEE architecture specification to infer the exception handling policy of the system. Second, Tomcat is a widely adopted open source project with source code and bug report repositories publicly available. So we could use EPL to check whether Tomcat implementation complies with the exception handling requirements documented in the JEE architecture specification and we could also check whether the violations observed were related to reported faults of the system. Exception handling faults of this system were collected in the study presented in Chapter 3. The study presented in Chapter 3 also collected faults from the Hadoop system. We did not use Hadoop in this study because this system did not have publicly available any type of documentation describing its architecture, nor any other structural aspect of the system. Thus, there was no sufficient information to infer an exception handling policy for Hadoop.

The JEE architecture specification is described in the Java Specification Request – JSR – number 342 (JSR-342). JSRs are descriptions and final specifications for the Java platform. They are developed by expert members affiliated to the Java Community Process (JCP), a community that comprise commercial, educational and non-profit organizations, as well as Java User Groups and individual Java users. After an initial proposal, JSRs are reviewed by the Java community and a JCP Executive Committee. After this review process, the leader of the experts group checks the reference implementation and the JSR specification before sending them to the Executive Committee for final approval. Once approved, the specification and reference implementation are published. JSRs are developed by Java experts and are only published after a thorough review and discussion process, so they can be trusted as reliable specifications of Java technologies.

We also used as target systems in this study the Mobile Media and Health Watcher systems. First, they are well documented systems that have been used in previous empirical studies that assessed their software architecture (ARCOVERDE et al., 2013, MACIA et al., 2012, MACIA et al., 2012a) and their exception handling implementation (CACHO et al., 2008, COELHO et al., 2008, SALES and COELHO, 2011). So there were sufficient information about these systems to infer their exception handling policies. Second, faults in these systems were assessed and reported in previous studies (BURROWS et al., 2010, FERRARI et al., 2010). Thus, we could check whether violations of the exception handling policies were related to these previously reported faults. When compared to Tomcat, Mobile Media and Health Watcher have less strict robustness requirements. Analyzing these systems allowed the investigation of EPL in scenarios in which programs did not favor robustness in their early versions, but are trying to improve it in later versions.

Data Collection and Analysis Method

In the context of Tomcat, we extracted exception handling-related requirements from its architectural specification in order to infer its exception handling policy. To extract these requirements from the architectural specification, we performed searches with keywords related to exception handling: *"exception"*, *"handling"*, *"catch"*, *"throw"*, *"raise"* and *"re-map"*. We identified five different requirements related to exception handling. From these five requirements, two were specified for modules implemented by Tomcat; the other three were specified for modules of the JEE architecture that are not implemented by Tomcat.

In the context of Mobile Media and Health Watcher, we relied on the extensive documentation produced in previous empirical studies to infer their exception handling policies. In particular, we relied on the documentation produced in previous empirical studies that assessed their software architecture (ARCOVERDE et al., 2013, MACIA et al., 2012, MACIA et al., 2012a) and their exception handling implementation (CACHO et al., 2008, COELHO et al., 2008, SALES and COELHO, 2011).

After producing the intended exception handling policy specification, we used EPL to automatically verify whether the source code of the target systems adhered or not to it. Then, we manually examined the policy violations to check whether they co-occurred in the same methods where exception handling faults were previously reported in the target systems. We also manually examined the methods where violations and faults co-occurred to check whether the causes of the violations were related to the causes of the faults.

4.4.2 Data Analysis

This section presents the analysis of the data gathered in this study. First, we present the analysis of the data collected in the context of Tomcat. Then, we present the analysis of the data collected in the context of Mobile Media and Health Watcher.

Tomcat Analysis

After inspecting the JEE architecture specification, we found two exception handling-related requirements for Tomcat. The first exception handling requirement specified for modules present in Tomcat is:

> ***Requirement 1:*** *The container must throw the javax.naming.OperationNotSupportedException from all the methods of the javax.naming.Context interface that modify the environment naming context and its subcontexts. (JSR-342, pg. 78)*

Notice that the previous exception handling requirement is expressed in terms of an obligation that specific source code elements have to comply. For the first requirement, specific methods of classes implementing the `javax.naming.Context` interface are obligated to raise the `OperationNotSupportException` type. This requirement is expressed in EPL by defining one compartment and one rule, as shown in the following code snippet:

```
define X.* as compartment CONTEXT where X is subtype of org.
   apache.naming.Context;
CONTEXT must raise javax.naming.OperationNotSupportedException;
```

The elements of interest related to the Requirement 1 were specified as the *CONTEXT* compartment, which was defined in terms of a subtype relation, as shown in the previous code snippet. Then, the rest of the exception handling requirement was specified with a single *Must* rule.

The second exception handling requirement extracted from the JSR-342 is the following:

> ***Requirement 2:*** *Web containers must throw a java.lang.IllegalArgumentException if an object that is not one of the above types, or*

another type supported by the container, is passed to the setAttribute or putValue methods of an HttpSession object corresponding to a Java EE distributable session. (JSR-342, pg. 174)

The second exception handling requirement also expresses an obligation, but in a different context and for a different exception. For the second requirement, the `setAttribute` and `putValue` methods of classes that implement the `HttpSession` interface are obligated to throw the `IllegalArgumentException` type. This requirement is also expressed in EPL with one compartment and one rule definition:

```
define X.setAttribute, X.putValue as compartment SERVLET-
   SESSION where X is subtype of javax.servlet.HttpSession;
SERVLET-SESSION must raise java.lang.IllegalArgumentException;
```

The elements of interest related to the second requirement were specified as the *SERVLET-SESSION* compartment, which was also defined in terms of a subtype relation. Then, the obligation imposed by the requirement was expressed by means of a single *Must* rule. It is worth mentioning that the feature in EPL for defining compartment in terms of subtype relations was not in the first version of the language; it actually emerged during the execution of this case study.

After specifying these requirements using EPL, we verified whether Tomcat's source code was adhering to the JEE specifications. We verified the implementation of Tomcat version 7.0.0. In this version, the rule specified for Requirement 2 was adhered to, whereas the rule for Requirement 1 was violated. From the faults collected in the study presented in Chapter 3, there was only one in the compartments specified. This fault was reported as a critical bug and was related to the violation of the Requirement 1.[1] This bug was reported on August 30 of 2011. One developer reported the following description for the bug:

The problem happens, if someone calls close() in the NamingContext object.

When developers called the `close` method in the `NamingContext` object, it raised an instance of `NamingException`. This caused a system failure by abruptly terminating the system execution. Developers provided a first fix on August 31 of 2011. However, on October 26 of 2011 the bug report was reopened. The developer who reopened the bug report mentioned that he had the same problem, but in another class implementing the Context interface:

[1]Available at: `https://bz.apache.org/bugzilla/show_bug.cgi?id=51744`

> *It appears that something is not quite right with this fix in 7.0.22. The following worked just fine in 7.0.14 (and GlassFish, WebLogic and WebSphere) and now fails on envCtx.close() with "Context is read only" message.*

On October 27 of 2011 one developer mentioned that those problems were related to Tomcat not adhering to the JEE specification. He first quoted the exception handling Requirement 1 in the bug report and then mentioned the following:

> *I would argue that the close() method is a method that "modifies the environment naming context" and therefore an exception should be thrown here. Tomcat is, however, not throwing the right exception in this case.*

Moreover, this bug received a new fix on October 28 of 2011. However, the same bug was once again reopened on June 21 of 2012 with the same problem being reported for other Tomcat versions. Developers mentioned:

> *I just installed 7.0.23 and I still see "Context is read only" exception thrown.*

> *I'm using Tomcat 7.0.25 and am still seeing this same issue.*

The bug report was finally closed on June 21 of 2012. The correct understanding of the intended use of exceptions in this specific context of Tomcat required discussions among 6 developers that lasted almost 10 months. Even after discovering the exception handling requirement related to the bug, developers faced difficulties in correcting the bug because it was repeated in different classes and different versions of the system. This may explain why the bug report was reopened twice and required new fixes in this period. Using EPL we created a short specification of the exception handling requirements defined for the JEE architecture and identified violations in the source code. These violations pointed directly to the causes of the reported bug. Therefore, Tomcat's verifiable exception handling policy was able to detect a severe fault in the exception handling code.

Mobile Media Analysis

We defined Mobile Media's exception handling policy aligned with its architecture. The Mobile Media architecture adheres to the Model-View-Controller (MVC) architecture pattern, so we defined one compartment for each module of the MVC pattern. Each compartment has the following responsibilities. The *Controller* compartment centralizes the exception handling by handling all exceptions. The *Model* module in Mobile Media's architecture comprises two sub-modules: *Domain*, responsible for abstracting the domain concepts, and *Data Access*, responsible for accessing persistence APIs. In the exception handling policy, *Data Access* is responsible for re-mapping API exceptions to application-defined exceptions. The *Domain* and *View* modules do not handle or raise any exceptions. The complete exception handling policy specification produced for Mobile Media v.9 comprised 5 compartments and 18 rules definitions.

From a total of 18 rules specified for the Mobile Media, 5 different rules were violated in the source code. Moreover, each rule could be violated more than once. The total number of violations observed in the source code is presented in Table 4.5.

Table 4.5: Mobile Media's Policy Violations

Site Type	Total	With Violations
Handling	63	27
Raising	9	2
Re-mapping	19	4

Table 4.5 presents the total number of handling, raising and re-mapping sites that exists in the source code, as well as the corresponding number of violations observed in each site type. A handling site is a method in the source code that handles an exception. A violation observed in a handling site means that a specific handling site violates a *Handle* rule. These definitions hold similarly for the other site types. A total of 33 violations were observed in Mobile Media. From this total, 27 violations were observed in handling sites, 4 in re-mapping sites and 2 in raising sites.

In addition, a total of 9 faults reported in the context of Mobile Media was collected from previous empirical studies (BURROWS et al., 2010, FERRARI et al., 2010). We manually reviewed these reports to check which faults were related to exception handling. From the total of 9 reported faults, 5 faults were related to exception handling.

Next, we further detail these violations and their relationship with exception handling faults, when existent.

Handling Site Violations. From the 27 violations in handling sites, 16 occurred in the *View* compartment, 6 in the *Domain* compartment and 5 in the *Data Access* compartment. These violations were related to exceptions being handled in the wrong place, since the *Controller* compartment was responsible for handling all exceptions. Moreover, there were 4 co-occurrences of policy violations and exception handling faults in the same method: 3 co-occurrences in the *Domain* compartment, 1 in the *Data Access* compartment and 1 in the *View* compartment. The following code snippet exemplifies one co-occurrence of a policy violation and an exception handling fault observed in the *View* compartment:

```
public void stopVideo() {
   try {
      if(player != null) player.stop();
   } catch(Exception e) {
      e.printStackTrace();
   }
}
```

The previous code snippet shows the `stopVideo` method extracted from the `PlayVideoScreen` class of Mobile Media. The policy violation in this method refers to the *catch* block declaring the generic exception type `Exception`, which it is not allowed to do. The `stop` method is an implementation of the interface `javax.microedition.media.Player` and may raise exceptions of the types `javax.microedition.media.MediaException` and `java.lang.IllegalStateException`. The documentation of the `stop` method specifies that exceptions of the `MediaException` type may be raised if the `Player` cannot be stopped. The documentation also specifies that exceptions of the `IllegalStateException` type may be raised if the `Player` is closed. The fault reported in this method occurred when a `MediaException` was raised. In this case, the generic *catch* block captured it and only printed its stack-trace. As a consequence, the player was not actually stopped and the user of the application was not informed about the problem. Then, the user of the application observed a failure. Therefore, not only the policy violation and the exception handling fault co-occurred in the same method, but the incorrect and generic *catch* block was the cause of both the policy violation and the exception handling fault. In this case, the `MediaException` was supposed to be propagated to the controller module of Mobile Media, which is responsible for centralizing exception handling.

It is worth mentioning that from the 16 handling site violations in the *View* compartment, 15 were related to generic *catch* blocks and 1 was related to a *catch* block declaring the `MediaException` type. Also, all these *catch* blocks were ignoring the captured exceptions, as depicted in the previous code snippet. This way, although only one of these violations co-occurred with a previously reported fault, it is possible that the other violations are related to active faults that were not reported in previous studies. Similarly, it is also possible that these violations are related to "dormant" faults, i.e., faults in the source code that did not lead to a subsequent failure only because their source code were not exercised yet.

Re-mapping Site and Raising Site Violations. There were 4 re-mapping site violations in Mobile Media: 3 in the *Domain* compartment and 1 in the *Data Access* compartment. There were also 2 raising site violations, both in the *View* compartment. None of the re-mapping and raising site violations co-occurred with exception handling faults.

Health Watcher Analysis

We also defined Health Watcher's exception handling policy aligned with its architecture. The Health Watcher is a web-based system for registering complaints about health units. The architecture of the Health Watcher system is structured into multiple tiers, in which each of the following modules correspond to one tier: *GUI*, *Business* and *Persistence*. It also has a module named *Façade* that manages the communication between *Business* and *GUI*.

In Health Watcher's exception handling policy, we defined one compartment for each one of the previous modules. Also, we defined the following responsibilities for each compartment:

- *Persistence* is responsible for re-mapping API-specific exceptions to the application-specific `PersistenceException` type. It is also allowed to raise instances of the `PersistenceException` type.
- *Business* is the only compartment allowed to raise exceptions of the `BusinessException` type. It is also responsible for propagating exceptions of the `PersistenceException` type to *Façade*.
- *Façade* is responsible for propagating exceptions of the `BusinessException` and `PersistenceException` types to *GUI*.
- *GUI* is responsible for handling instances of the `BusinessException` and `PersistenceException` types.

Table 4.6: Health Watcher's Policy Violations

Site Type	Total	With Violations
Handling	197	26
Raising	99	65
Re-mapping	185	18

The complete exception handling policy specification produced for Health Watcher (version 10) comprised 4 compartments and 13 rules. From a total of 13 rules specified, 7 different rules were violated in the source code. Table 4.6 details the policy violations observed in Health Watcher.

A total of 109 violations were observed in Health Watcher. From this total, 65 were raising site violations, 26 were handling site violations and 18 were re-mapping site violations.

In addition, a total of 13 faults reported in the context of Health Watcher was collected from previous empirical studies (BURROWS et al., 2010, FERRARI et al., 2010). We manually reviewed these reports to check which faults were related to exception handling. From the total of 13 reported faults, 6 faults were related to exception handling.

Next, we further detail these violations and their relationship with exception handling faults, when existent.

Raising Site Violations. From the 65 raising site violations in Health Watcher, 64 occurred in the *Persistence* compartment and 1 in the *GUI* compartment. All raising site violations were related to methods raising an exception that should have been raised by the methods in the *Business* compartment. Among the exception handling faults reported in Health Watcher, there was no fault related to the raising of exceptions. Therefore, no co-occurrence of policy violations and exception handling faults in raising sites was observed in the context of Health Watcher.

Handling Site Violations. From the 26 handling site violations in Health Watcher, there were 20 in the *GUI* compartment. All the handling site violations in the *GUI* compartment were related to *catch* blocks declaring the generic type `Exception`. None of these violations co-occurred with previously reported exception handling faults.

Moreover, there were 6 in the *Persistence* compartment. All the handling site violations in the *Persistence* compartment were related to *catch* blocks capturing exceptions that should have been captured by methods in the

Business compartment. From the 6 handling site violations in *Persistence*, 2 violations co-occurred with previously reported exception handling faults. The following simplified code snippet exemplifies one of the co-occurrences observed in the *Persistence* compartment:

```
void insertAnimal(Complaint complaint){
    if (complaint.getAddress() != null) {
        try {
            insert(getAddress());
        } catch (ObjectNotValidException e){ }
    }
    insertAnimalComplaint(complaint);
}
```

The `insertAnimal` method in the previous code snippet implements the logic that persists a complaint related to a given animal. First, if the complaint has a valid address associated, it persists this address. Then, it persists the complaint itself. If an `ObjectNotValidException` occurs when trying to persist the address, then a *catch* block captures the exception and ignores it with an empty catch block. Once the exception is handled, the normal execution continues and the `insertAnimalComplaint` method is invoked.

When an `ObjectNotValidException` occurred while trying to persist the address, the address was not persisted and the `insertAnimal` continued its normal execution and persisted the complaint. That is, the complaint was persisted in the system, but the corresponding address was not. This way, ignoring the exception with an empty *catch* block caused a failure.

Re-mapping Site Violations. From the 18 re-mapping site violations in Health Watcher, 14 occurred in the *Business* compartment and 4 in the *Persistence* compartment. Moreover, from the 14 re-mapping site violations in the *Business* compartment, 2 co-occurred with previously reported exception handling faults. No violation in the *Persistence* compartment co-occurred with previously reported exception handling faults. The 2 co-occurrences of policy violations and exception handling faults in the *Business* were related to checked exceptions being re-mapped to unchecked exceptions. The following simplified code snippet exemplifies the co-occurrence of exception handling faults and policy violations observed in the *Business* compartment:

```
private long writeTimeStamp(String tableName, String id) throws
    PersistenceException{
   try {
      String sql = createSql(tableName, id);
      PersistenceMechanism.executeQuery(sql);
      (...)
```

```
        return answer;
    } catch (Exception ex) {
        ex.printStackTrace();
        throw new RuntimeException(ex);
    }
}
```

In the previous code snippet, the `writeTimeStamp` method is in the *Business* compartment. The policy violation in this method refers to the re-mapping from the `Exception` type to the `RuntimeException` type, which it is not allowed to perform. The `searchTimeStamp` method retrieves a list of complaints for a given health unit registered in Health Watcher. To do so, it invokes the `executeQuery` method in the *Persistence* compartment. The `executeQuery` method may raise exceptions of the `SQLException` type. When these exception are raised, they are captured by subsumption by the generic *catch* block implemented in the `writeTimeStamp` method. Then, the captured exceptions are re-mapped to the `RuntimeException` type. The re-mapped exceptions flow through the boundaries of the `writeTimeStamp` method and are not captured by any method in Health Watcher, causing the system termination due uncaught exceptions. Therefore, the incorrect re-mapping performed is the cause of the policy violation and also of the fault in the `writeTimeStamp` method.

The violation in the previous example seemed to be induced by another violation in the same call-chain. The `executeQuery` method was expected to re-map within its boundaries the `SQLException` to the `PersistenceException` type. Then, the `writeTimeStamp` method was expected to only propagate the `PersistenceException`. However, since the `writeTimeStamp` method received a `SQLException`, it would not be possible to propagate this exception without modifying the exceptional interface of the method. Thus, the developer might have opted to propagate the exception in a easy way by simply re-mapping the `SQLException` to the `RuntimeException` type. Thus, it might be the case that violations in the source code actually induce developers introducing other violations. We observed a total of 17 call-chains containing more than one violation in Health Watcher.

The other 2 re-mapping violations in *Business* were also related to checked exceptions being re-mapped to unchecked exceptions. Although these violation did not co-occur with previously reported faults, it is still possible that these violations lead to subsequent failures, since there is no handler for the `RuntimeException`. It might be the case that these violations were still not exercised during program execution and remain dormant.

4.4.3 Results and Discussions

This section presents the results of this study, as well as the discussion of the implications of these results. First, the results that directly address the questions of this study are presented. Next, other results and findings are discussed.

Violations Point to the Causes of Exception Handling Faults

The co-occurrence analysis of the collected violations and exception handling faults supports the answer to the questions of this study:

> *Do violations detected by EPL and exception handling faults co-occur in the same methods?*
>
> *Are the causes of violations detected by EPL related to the causes of exception handling faults?*

In this study, we observed that some policy violations co-occurred with exception handling faults in methods of the target systems. In addition, we also observed that in all co-occurrences of policy violations and exception handling faults their causes were related. More specifically, the causes of the policy violations were the same causes of the exception handling faults. In Tomcat, a raising site violation was related to an “Incorrect raiser” fault. In Mobile Media and Health Watcher, there were handling site violations related to “Incorrect handler” faults. And in Health Watcher, there were also re-mapping site violations related to “Incorrect re-mapper” faults. Therefore, policy violations and exception handling faults co-occurred in the same methods in the target systems analyzed and the causes of the policy violations were the same causes of the reported faults.

Unrelated Faults and Policy Violations

Exception handling policy specifications define high-level decisions on exception handling design that must be adhered in the source code. Policy violations refer to the parts of the source code that deviate from the intended exception handling implementation. And these violations may point to potential exception handling faults in the source code, as previously discussed. However, there were reported exception handling faults that were not related to any policy violations. For example, one exception handling fault in Mobile Media was described as follows:

> *Ignoring exception RecordStoreNotFoundException. This is happening when deleting all existing albums from record stores. The application does not support 0 record stores and automatically creates the default album.*

This fault in Mobile Media is related to low-level implementation details, i.e., it is related to the lack of handling actions that should have been implemented. Since policy specifications in EPL are focused on high-level design decisions (e.g., which module is responsible for handling a given exception), low-level implementation details are beyond the scope of EPL. Consequently, policy violations cannot assist the identification of exception handling faults related to low-level implementation details.

It is worth highlighting that not supporting the specification of low-level implementation details was an intentional decision in the language design, rather than an implementation limitation. We intentionally designed EPL not to support the definition of low-level implementation details because specifying these details would break the information hiding principle of modules. Also, specifying this kind of information would make our language too complex, given the high number of possible details related to the implementation of exception handling. For example, one exception handler can log the exception, release pre-allocated resources and then shut down the system. Specifying which handling actions should be taken, how they are implemented and in which order, would probably hinder one of the main design goals of domain-specific languages, which is to keep the language concise. Participants of our user-centric study appreciated the simplicity of EPL, so making the language more complex could possibly have a negative impact on developers acceptance towards the language.

There were also policy violations that were not related to reported faults. We hypothesize the reasons why violations were not related to faults. For Health Watcher and Mobile Media, there were few exception handling faults reported, so it might be the case there were other unreported faults co-occurring with violations, but we were not aware of the localization of these faults. On the other hand, for Tomcat, there was little information about its exception handling policy, so we may have missed violations in the source code. Finally, the violations detected in the source code may actually not be related to any fault. Yet, we believe that developers must treat each violation as potential threats to software robustness and, therefore, they should be detected and repaired.

4.4.4 Threats to Validity

This section discusses the study limitations based on the threats to the study validity, presenting the measures took to mitigate these threats.

Construct Validity

A first threat to the construct validity of this study concerns possible biases in the ground truth of the exception handling faults used. To mitigate this, we relied on exception handling faults that were reported in previous empirical studies conducted by other researchers, in the case of Mobile Media and Health Watcher, and in exception handling faults reported by the system's users and developers, in the case of Tomcat. Another threat concerns possible biases in the exception handling policies produced. We limited this threat by producing the exception handling policy for Tomcat based on its architectural specification. Similarly, we produced the exception handling policies for Mobile Media and Health Watcher aligned with their intended software architecture, as described in previous empirical studies.

Internal Validity

Threats to the internal validity of this study refer to the limitations of the *EPL Verifier*. As we discussed in Section 4.2, the type-inference algorithm implemented by the *EPL Verifier* to determine the type of the raised exception may produce imprecise results in specific scenarios. In particular, the type-inference algorithm implemented by the *EPL Verifier* may produce imprecise results when the type of the raised exception cannot be precisely determined statically either because the type of the raised exceptions is inferred from a conditional expression or from the returned type of a virtual method invocation. For this reason, the type-inference algorithm implemented by the *EPL Verifier* may interfere in the results related to raising and re-mapping site violations. To assess to what extent the use of the *EPL Verifier* interfered the results discussed, we assessed in how many cases the type-inference algorithm produced imprecise results.

For the Mobile Media and Health Watcher target systems, there were no cases in which the *EPL Verifier* found conditional expressions or method invocations during its analyses. Therefore, the use of the *EPL Verifier* did not interfere in the results discussed for these target systems. For the Tomcat target system, all the *throw* statements within the compartments analyzed referred to new instance creation expressions. Therefore, the type-inference algorithm also did not introduce any imprecision in the analysis presented.

Even so, we analyzed the source code of Tomcat to identify the cases in which the *EPL Verifier* would have produced imprecise results. From the total of 1880 *throw* statements in Tomcat, in 1582 cases the type of the raised exceptions were inferred from new instance creation expressions, 152 cases were inferred from class cast expressions, 128 cases were inferred from references to arguments of *catch* blocks, 17 case were inferred from virtual method invocations and 1 case was inferred from a conditional expression. We manually inspected the cases where the type of the raised exception was inferred from virtual method invocations or conditional expressions to assess if the *EPL Verifier* would have interfered in the analyses if these *throw* statements were within the compartments analyzed. From the 17 cases where the *throw* statement referred to virtual method invocations, 1 referred to an API method invocation, so we could not inspect its source code. For the other 16 virtual methods declared in the application, we inspected their source code and observed that all of them returned the same type declared in the method signature. Therefore, the analysis of virtual method declarations would not have interfered in the analysis.

In the context of the Tomcat target system, there was only one *throw* statement referring to a conditional expression, as shown in the following code snippet:

```
if (t instanceof InvocationTargetException) {
  InvocationTargetException i=(InvocationTargetException)t;
  throw i.getCause() != null ? i.getCause() : i;
}
```

In the previous code snippet, the *throw* statement refers to a conditional expression. The *then* expression of the conditional expression refers to an invocation to the `getCause` method, whereas the *else* expression refers to the variable name `it`. The return type of the `getCause` method is the `Throwable` type and the type of the variable `it` is the `InvocationTargetException` type, which is inferred from the class cast expression. The *EPL Verifier* considers that the previous code snippet raises the `Throwable` and the `InvocationTargetException` types. In the previous code snippet, the possible imprecision stem from the fact that the runtime type of the object returned by the `getCause` may be a subtype of the `Throwable` type. However, the analysis performed by the *EPL Verifier* produces the same results produced by the Java compiler. For the previous code snippet, the Java compiler requires that the `Throwable` type is either handled locally, or declared in the method's exceptional interface, regardless of the exact type of the object returned by the `getCause` method. Thus, the analysis performed by the *EPL Verifier* for *throw* statements referring to method invocations is as conservative as the analysis performed by the Java compiler.

External Validity

The main threat to external validity is related to the number of target systems analyzed. To minimize these threats, we tried to use systems from different natures: Health Watcher is a web-based system, Mobile Media is a mobile application and Tomcat is a web-server. This way, we tried to use systems with different software architectures and with different exception handling policies. Even so, we are aware that more studies involving a higher number of systems should be performed in the future to promote the generalizability of the results observed in this study.

4.5 Related Work

The proposed domain-specific language is a means for explicitly specifying and automatically verifying exception handling policies. Similarly, solutions aimed at assuring architecture quality by specifying and verifying architectural design rules have been vastly explored in the software architecture community in the last years. According to Knodel and Popescu (KNODEL and POPESCU, 2007) and Van Ommering, Krikhaar and Feijs (VAN OMMERING et al., 2001), these architectural solutions can be divided in three main categories: (i) Reflexion Models, (ii) Relation Conformance Rules and (iii) Component Access Models.

Reflexion Models compare high-level descriptions of the intended architecture of a system with its source code to detect divergences and absences. Divergences occur when relations not prescribed in the intended architecture exist in the source code, whereas absences occur when relations prescribed in the intended architecture do not exist in the source code. Solutions based on Relation Conformance Rules specify design rules that express allowed or forbidden relations between architectural elements. Finally, Component Access Models solutions specify components interaction by means of specifying components provided and required ports, as well as connection between ports. These solutions are inspired by Architecture Description Languages (CLEMENTS, 1996). Among these three categories, our proposed solution has more similarities with those based on relation conformance rules. Table 4.7 presents a comparison of EPL with related works based on Relation Conformance Rules (ABRANTES and COELHO, 2015, EICHBERG et al., 2008, GURGEL et al., 2014, SALES and COELHO, 2011, TERRA and VALENTE, 2009).

Table 4.7: Comparison of EPL with Related Works

Solutions	**Semantics of Rule Types**	**Supported Exception Handling Dependencies**
EPL	Obligation, Permission, Prohibition	Handle, Propagate, Raise, Re-map, Re-throw
Abrantes and Coelho (2015)	Obligation	Handle, Raise
Cacho et al. (2008)	Obligation	Handle, Propagate, Raise
Eichberg et al. (2008)	Obligation	Handle, Raise
Gurgel et al. (2014)	Obligation, Permission, Prohibition	Handle
Sales and Coelho (2011)	Obligation	Handle, Raise
Silva and Castor (2013)	Obligation	Handle, Propagate, Raise
Terra and Valente (2009)	Obligation, Permission, Prohibition	Raise

As can be observed in Table 4.7, in terms of the semantics of the provided rule types, EPL and the solutions proposed by Gurgel et al. (GURGEL et al., 2014) and by Terra and Valente (TERRA and VALENTE, 2009) provide rule types with the semantics of permission, prohibition and obligation. The other solutions provide only rule types with the semantics of obligations.

In terms of the supported exception handling dependencies, EPL is the only solution that supports all the “canonical” dependencies between exceptions and code elements described in Chapter 2. The other solutions only support the *Handle*, *Propagate* and *Raise* dependency. When compared to these other solutions, the main contribution of EPL is to provide a wider vocabulary of exception handling dependencies to specify and verify exception handling policies. Next, we detail other similarities and differences between EPL and the other solutions.

The solution proposed by Abrantes and Coelho (ABRANTES and COELHO, 2015) and by Sales and Coelho (SALES and COELHO, 2011) are the most similar to ours. Their solution is aimed at specifying exception handling contracts, although they do not support all the “canonical” exception handling dependencies. An exceptional contract specifies an intended exception propagation path, i.e., it specifies the specific places in the source code where specific exceptions are raised and handled, and also which specific exception types may flow between these places. Thus, their solution specifies exception handling policies in a level closer to the implementation level, whereas our solution specifies policies in a level closer to the design level.

Another major difference between our solutions is how we check the source code conformance: we check it statically, while they check it dynamically. Abrantes and Coelho use their exceptional contracts to monitor the execution of a program. Sales and Coelho generate from their exceptional contracts partial JUnit test cases to stimulate the exceptional behavior of the system. Moreover, the solution proposed by Sales and Coelho requires the intervention of developers to finish the implementation of the partially generated test cases. In fact, testing exception handling code often requires extra effort of developers. It is often difficult to set-up test scenarios that break the assertions associated to exceptional conditions in software modules. Therefore, the use of fault-injection mechanisms is often required to actually exercise the exception handling code. Our solution is based on static analysis because developers can detect exception handling violations in the source code early in the development process, preventing them from remaining dormant in the source code and cause failures. Moreover, the *EPL Verifier* requires only the exception handling policy specification and the system source code. And writing policies can be assisted by mining exception handling rules from source code (THUMMALAPENTA and XIE, 2009).

Although we have favored a solution based on static analysis, we believe that it may be used in collaboration with dynamic analysis. In addition to using the *EPL Verifier* to statically check the source code conformance, we could extend the EPL tool apparatus to also generate partial test cases in order to dynamically check the conformance of the source code in exceptional scenarios that require more specific implementation details and, therefore, cannot be described on a system-level specification language such as EPL. Similarly, policy specifications in EPL could also be used to monitor the execution of programs.

The solutions proposed by Eichberg et al. (EICHBERG et al., 2008), Gurgel et al. (GURGEL et al., 2014) and Terra and Valente (TERRA and VALENTE, 2009) specify architectural design rules aimed at detecting and preventing architectural degradation problems. For this reason, these solutions focus on expressing architectural design rules in terms of dependencies originated from source code elements accessing methods and fields, instantiating new class instances, extending classes, implementing interfaces, among others. Not all exception handling dependency are supported by their solutions. Eichberg et al.considers the *Handle* and *Raise* dependencies as the generic dependency named *Use*; Gurgel et al. supports only the *Handle* relation, whereas Terra and Valente supports only the *Raise* relation.

Similarly to specifications in EPL, which are expressed in terms of compartments, the solutions proposed by Eichberg et al., Gurgel et al.and Terra and Valente are also defined in terms of abstractions that group together elements of interest at the system implementation level. These abstractions are also defined in terms of name patterns and subtype relations, just like the definition of compartments in EPL. The solution proposed by Eichberg et al. is the only solution that allows expressing design rules in terms of dependencies combined with logic operators for conjunction, disjunction and negation. All the other solutions, EPL inclusive, may only express their rules in terms of atomic dependencies. Finally, the solution proposed by Gurgel et al. is the only one that provides a compositional mechanism that allows the specialization and reuse of abstract design rules in the context of different projects. So far, we did not find evidences that the specification of exception handling policies requires the combination of exception handling dependencies with logic operators nor requires the specialization and reuse of abstract rules in different projects. Still, these are investigation paths that we might explore in the near future as possible improvements in EPL.

In the exception handling literature, there are works that extend exception handling mechanims of programming languages to support the explicit specification of exception handling rules in the source code (CACHO et al., 2008, SILVA and CASTOR, 2013). Cacho et al. (CACHO et al., 2008) extended the exception handling mechanisms of AspectJ, whereas Silva and Castor (SILVA and CASTOR, 2013) extended the mechanisms of Java, to provide new language constructs to specify and verify the places in the source code where exceptions are expected to be raised, propagated and handled. These approaches mainly differ from ours because they specify parts of the exception handling policy with the own programming language, whereas our approach uses a domain-specific language. In this sense, when compared to our solution, the solutions proposed by Cacho et al.and Silva and Castor have the advantage of not requiring that developers learn a new language, although developers still have to learn a few new language constructs. In order to ease the learning and use of EPL, we designed it with a concise vocabulary of terms similar to those used in exception handling mechanisms in programming languages. We also designed it to produce readable specifications. The observations of our user-centric study suggests that EPL is indeed easy to learn and use, although more rigorous studies are needed to confirm that.

The solutions proposed by Cacho et al. and Silva and Castor have the main limitation of verifying only parts of the system that are implemented in the programming language that they used to express their exception handling rules. In multi-language systems, where exceptions may flow from a module implemented in one language to a module implemented in another language, the specified exception handling rules cannot be completely verified. So far, EPL also has this limitation, since its verifier is implemented only for Java. But since EPL is a specification language agnostic of programming language, we plan as future work to implement EPL for other programming languages and start to investigate how exceptions flow between modules implemented in different programming languages and if exception handling rules are violated in these scenarios.

Finally, in the exception handling literature there are also a few efforts to support developers in properly designing exception handling of software systems. Litke (LITKE, 1999) proposed a method to design fault tolerant Ada systems. Litke's method proposes exhaustive specification of exceptions at modules boundaries by enumerating and defining the semantics of all exceptions that cross these boundaries. Then, the method recommends the automated verification of appropriated handlers for each exception specified in module boundaries.

Robillard and Murphy (ROBILLARD and MURPHY, 2000) proposed a method to design robust Java systems by adapting Litke's method. These methods provide good methodology for specifying exception handling in module boundaries, but both lack an explicit definition of the intended exception handling policy. They also lack tool support. Consequently, there is no way to automatically check the source code conformance to the intended exception handling policy.

Malayeri and Aldrich (MALAYERI and ALDRICH, 2006) extended Java to support the specification and verification of exceptions at module boundaries, as proposed by the method of Robillard and Murphy. Malayeri and Aldrich specify and verify exceptional interfaces at the module level, instead of at the method level, as performed by the Java compiler. However, exceptional interfaces of modules specify only which exception can traverse their boundaries. There is no way to express exception handling responsibilities that comprise exception handling policies. Therefore, these solutions do not provide proper support for specifying and automatically verifying exception handling policies. The works from Litke and Robillard and Murphy provide methods that could be adapted to assist developers during the specification of their exception handling policies. Thus, we plan to elaborate guidelines to help developers in how to specify their exception handling policies.

4.6 Summary

In this chapter, we presented the second contribution of this thesis: EPL, a domain-specific language to specify and verify exception handling policies (Sections 4.1 and 4.2). The definition and enforcement of exception handling policies allow the detection of exception handling violations in the source code, achieving the first part of the goal of this thesis. All the results of this chapter were reported in a paper accepted to be published (BARBOSA et al., 2015).

With our user-centric observational study (Section 4.3), we could better understand the trade-offs related to different language design decisions based on concrete and well-documented observations and experiences reported by participants. We observed some language characteristics that hindered the definition of exception handling policies. These observations motivated us to add new language constructs to EPL. In addition, the participants of our user-centric study recognized the importance of having explicit exception handling policies in their projects. They also considered exception handling policies expressed in EPL useful to support quality assurance practices. This was the main reason why participants affirmed that they would adopt the proposed language in their activities. Participants also considered the language easy to learn and to have potential to improve their performance and productivity, although more rigorous studies must be conducted in the future to confirm real gains in performance and productivity.

The results of our case study (Section 4.4) revealed that violations of exception handling policies could help to directly detect causes of exception handling-related failures. This could not be detected with the basic verification performed by current exception handling mechanisms, such as those performed by the Java compile for the checked exceptions. In addition, our results also showed that the benefits of using our proposed language could be achieved even when only parts of a system are specified and verified, such as we did with Tomcat. There might be cases where specifying the exception handling policy for the whole system is not practical. For example, the architect or the lead developer might not have time to specify the whole system; he might have time to specify only small, but critical, parts of the system.

Finally, both designers and developers can benefit from our specification language for exception handling policies. Designers have at their disposal a succinct and expressive language to explicitly define their intentions regarding the use of exceptions within software projects. And with an explicit definition about the intended use of exceptions, developers can readily consult the specification to comprehend how they are supposed to maintain exception handling code. Both designers and developers can use the static analyzer to detect violations in the source code. Once these violations are detected, they must repair them. In the next chapter we present a recommender heuristic strategy aimed at supporting the repair of exception handling violations.

5 Repairing Violations in Exception Handling

With the EPL language presented in the previous chapter, developers can define their exception handling policies and detect exception handling violations in the source code. After detecting exception handling violations, developers must repair them to avoid potential failures caused by these violations. However, repairing exception handling violations is a difficult and error-prone task, specially when these violations are related to global exceptions. First, each repair requires understanding how exception handling should be implemented, so that the source code where violations are located can be modified to a version adhering to the intended implementation. Second, modifications in the exception handling code usually require performing changes in different parts of the program.

In fact, due to the inherent global nature of exceptions, changes in the exception handling code often have side-effects in apparently unrelated parts of the system. This way, while developers try to repair existing violations, they can easily introduce others in the source code of the system. And these violations may remain dormant in the source code until they are later exercised during runtime, possibly leading to subsequent failures. Given the inherent complexity in repairing exception handling violations, developers need proper assistance to perform this task.

In this context, this chapter presents the proposed solution that addresses the second research question of this thesis:

> ***RQ2.*** *How to support the repair of exception handling violations in the source code?*

In order to address the research question RQ2, we proposed RAVEN – a heuristic recommender strategy for supporting the repair of exception handling violations.[1] RAVEN is the proposed solution that complements the work of EPL. Once violations are detected, RAVEN can be employed to support their

[1]RAVEN is a loose acronym for **R**ep**A**iring **V**iolations in **E**xception ha**N**dling.

repair. Thus, by combining EPL and RAVEN, developers have proper support for detecting and repairing exception handling violations, fulfilling the research goal of this thesis.

Repairing exception handling violations requires global reasoning about the impact that modifications in the exception handling have. For this reason, the RAVEN strategy takes into account the global context of where exception handling violations occur to provide global-aware recommendations. In the case study presented in the previous chapter, we observed that some exception handling violations co-occurred in the same call-chain (Section 4.4.2). And violations in the same call-chain seemed to be related. For this reason, RAVEN analyzes the source code structure of all methods in the call-chain where violations are located. RAVEN uses information extracted from source code to build the solution space from where it constructs its recommendations. In addition, when policy specifications are available, RAVEN uses them to adjust its solution space in an attempt to produce more relevant recommendations. Each recommendation produced by RAVEN consists of a sequence of modifications that should be performed in the whole call-chain of methods where an exception handling violation occurs. The recommended modifications serve as a detailed blueprint of how exception handling violations can be removed from the source code.

The RAVEN strategy was analyzed in an experiment that evaluated its effectiveness. The effectiveness of RAVEN relates to its ability in producing recommendations able to repair exception handling faults. Moreover, the effects of using policy specifications in the effectiveness of RAVEN was also analyzed. The results of this experiment revealed that RAVEN produced relevant recommendations in approximately 70% of the cases. When policy specifications were available, it produced relevant recommendations in 97% of the cases. The results also showed that the benefits of using policy specifications to produce recommendations could be achieved with partial specifications. Therefore, development teams may benefit from the proposed recommender strategy, even when exception handling policies are only partially documented in their projects.

The rest of this chapter is structured as follows. Section 5.1 presents the proposed recommender heuristic and Section 5.2 describes the procedure conducted to evaluate this heuristic. Section 5.3 analyzes the data collected in the evaluation procedure, Section 5.4 presents the study results and Section 5.5 presents and discusses the threats to the study validity and the measures taken to mitigate them. Section 5.6 compares the RAVEN strategy to related works and Section 5.7 summarizes and concludes this chapter.

5.1 The RAVEN Strategy

This section details RAVEN, a recommender heuristic strategy that produces recommendations on how to repair exception handling violations. As previously discussed, repairing exception handling violations is a complex task that requires reasoning about the global context of exceptions. For this reason, RAVEN takes into account the global context of where exception handling violations occur to be aware of the impact that exceptions might have. In particular, given a method m where a violation occurs, the RAVEN strategy analyzes the whole call-chain of m to produce its recommendations. A recommendation produced by RAVEN is a sequence of modifications to be performed in the call-chain of m in order to repair the violation that occurs in m.

The RAVEN strategy comprises three steps. In the first step, the solution space where RAVEN produces its recommendations is built. This step is further explained in Section 5.1.1. In the second step, the solution space is traversed for constructing valid recommendations. This step is detailed in Section 5.1.2. Finally, in the third step, the valid recommendations are ranked. This step is detailed in Section 5.1.3.

5.1.1 Solution Space Construction

The first step in the RAVEN strategy is the solution space construction. The solution spaces constructed by RAVEN are defined as follows. Let $\mathbb{M}$ be the set of all methods in the system being implemented, $\mathbb{D}$ be the set of exception handling dependencies (*Handle*, *Raise*, *Propagate*, *Re-map* and *Re-throw*, as described in Chapter 2) and $\mathbb{T}$ be the set of all exception types in the system being implemented. Then, given a method m where a violation occurs, the solution space for this method – $\mathbb{S}(m)$ – is defined as:

$$\mathbb{S}(m) = \{(x, y, z) \in \mathbb{M} \times \mathbb{D} \times \mathbb{T} : x \in \mathbb{C}_m\}$$

where $\mathbb{C}_m$ is the set of methods in the call-chain of m (m inclusive). A solution space constructed by RAVEN comprises the tuples (x, y, z) that represent all the exception handling dependencies y that the methods x in the call-chain of m may establish with the exception types z.

The information used to construct the solution space is extracted from the source code of the system. RAVEN analyzes the source code of the methods in the call-chain of m and extracts information related to the exception handling dependencies already implemented by these methods. In addition, RAVEN complements this information by leveraging on functional similarity between methods. To do so, it is assumed that methods with similar functionalities use exceptions similarly. In other words, it is assumed that if a given method m establishes a given exception handling dependency d with a given exception type t, then methods that are functionally similar to m are likely to also establish a dependency d with t. This was inspired by source code-based recommender systems for software engineering that employ functional similarity to search for examples of APIs use (HOLMES and MURPHY, 2005,SAHAVECHAPHAN and CLAYPOOL, 2006,THUMMALAPENTA and XIE, 2007). It is worth making it clear that RAVEN does not construct its solution spaces by exhausting all possible combinations of exception handling dependencies and exception types that the methods in the call-chain of m may establish. The set of all exception types $\mathbb{T}$ comprises not only the exception types defined in the context of the system being implemented, but also the exceptions defined by third-party libraries. Given the large number of possibilities, we opted to construct the solution spaces by leveraging on functional similarity between methods.

To measure functional similarity between methods, we relied on the combination of similarity metrics that presented the best results in a previous empirical study performed by Higo and Kusumoto (HIGO and KUSUMOTO, 2014). In particular, we measured functional similarity (FS) between two methods by combining measures for their structural similarity (SS) and their vocabulary similarity (VS). The measurement for functional similarity relies on the assumption that if two methods have similar internal structures and they use similar sets of terms, then these methods are likely to have similar functions. The functional similarity between two methods m_1 and m_2 is computed as follows:

$$FS(m_1, m_2) = SS(m_1, m_2) + VS(m_1, m_2)$$

The measurement for structural similarity quantifies the similarity between the internal syntactic structures of two methods. In the procedure adopted by Higo and Kusumoto, the structural similarity between two methods m_1 and m_2 is measured by first producing token sequences S_1 and S_2 for each method. A token sequence S_i is obtained from the declaration of a method m_i by replacing all tokens representing variable names, method names and type names with special tokens. Keywords are preserved and white spaces, tabs and new line characters are discarded. Thus, the syntactic structure of a method is expressed in a simple yet representative string-based notation. After producing the token sequences S_1 and S_2, the longest common subsequence (LCS) between these sequences is computed. Finally, the structural similarity is measured as:

$$SS(m_1, m_2) = min\left(\frac{|LCS(S_1, S_2)|}{|S_1|}, \frac{|LCS(S_1, S_2)|}{|S_2|}\right)$$

where $|S_1|$ and $|S_2|$ represent the number of tokens in the token sequences S_1 and S_2, respectively, and $|LCS|$ represents the number of tokens in the longest common subsequence between the two token sequences. The higher the value of SS is, the more similar are the internal structure of the two methods.

The measurement for vocabulary similarity quantifies how similar are the sets of terms used by two methods. The vocabulary similarity between two methods is measured by first producing the sets of vocabulary terms for each method. Higo and Kusumoto considered the set of terms V_i of a given method m_i as the set of variable names, type names and method names invoked in the context of m_i. In addition to those terms used by Higo and Kusumoto, we also considered as part of the set of terms of a method the names and types of the parameters and the exception types declared in the signature of the method. Thus, we take into account not only the vocabulary of terms used within m_i, but also the terms used in its signature, particularly the exception types declared in its exceptional interface. Then, the vocabulary similarity between two methods m_1 and m_2 is measured by computing the Jaccard index between the sets V_1 and V_2, as shown in the next equation:

$$VS(m_A, m_B) = \frac{|V_A \cap V_B|}{|V_A \cup V_B|}$$

As shown in the previous equation, the vocabulary similarity between two methods is defined as the size of the intersection divided by the size of the union of the sets V_1 and V_2. The higher the value of VS is, the more similar the vocabulary of terms of two methods are.

In this context, assuming that methods with similar functionalities use exceptions in similar manners and considering a method m where a violation occurs, the solution space for this method is constructed as follows. For each method c_i in the call-chain $\mathbb{C}_m$ of the method m, first extract from the source code of c_i the exception handling dependencies d that it implements and the respective exception types t; then add the tuples (c_i, d, t) to the solution space. Next, search for methods that are functionally similar to c_i and let $Sim(c_i)$ be the set of methods that are functionally similar to c_i. Then, for each method s_j in $Sim(c_i)$, analyze the source code of s_j and extract the exception handling dependencies that it implements and the respective exception types. Let $Tup(s_j)$ be the set of tuples that represent the exception handling dependencies extracted from the source code of the method s_j. Then, for each tuple (x, y, z) in $Tup(s_j)$, add (c_i, y, z) to the solution space. Notice that if the source code of the method c_i is analyzed, the tuple (c_i, y, z) is not necessarily extracted from it. The exception handling dependency y with the exception type z was originally extracted from the method s_j, which is similar to c_i. Thus, functional similarity between methods allows expanding the solution space with new tuples.

In addition to expanding its solution space by leveraging on functional similarities between methods, RAVEN also adjusts its solution space when policy specifications are available. Exception handling policies define the intended exception handling dependencies that modules of a system are supposed to implement with exception types. For example, according to the exception handling policy of Health Watcher, the *GUI* tier is only allowed to handle instances of the `BusinessException` and `PersistenceException` types; handling exceptions of another type would be considered as violations of the intended policy. In this case, for all methods m that are part of the *GUI* tier, RAVEN prunes from its solution space any existing tuple of the form $(m, Handle, t)$, where t is an exception type different from `BusinessException` and `PersistenceException`. In addition, since methods that are part of the *GUI* tier are allowed to handle instances of the `BusinessException` and `PersistenceException` types, RAVEN also adds to its solution space tuples of the form $(m, Handle, BusinessException)$ and $(m, Handle,\ PersistenceException)$. In summary, when policy specifications are available, RAVEN adjusts its solution space by pruning prohibited tuples and by adding allowed tuples. This way, the solution space contains only policy-compliant tuples.

5.1.2 Constructing Recommendations

The second step in the RAVEN strategy is the construction of its recommendations. RAVEN starts constructing its recommendations by computing valid exception propagation paths that may be implemented in the context of the call-chain of a method. Given a method m where a violation occurs, RAVEN traverses the solution space of m to construct valid exception propagation paths in the context of the call-chain of m. A valid exception propagation path $\mathbb{P}$ produced by RAVEN has the following structure:

$$\mathbb{P} = \underbrace{(x_1, \mathit{Raise}, z_1)}_{\text{Raising Site}} \cdot \underbrace{(x_i, y_j, z_k)^*}_{\text{Intermediate Site}} \cdot \underbrace{(x_n, \mathit{Handle}, z_m)}_{\text{Handling Site}}$$

A valid exception propagation path $\mathbb{P}$ produced by RAVEN is a sequence of tuples that represents an exception propagation path from a raising site to a handling site, possibly passing through intermediate sites that either *Propagate*, *Re-throw* or *Re-map* the exception. These paths are constructed by exploring the solution space $\mathbb{S}(m)$ of the method m with a backtracking algorithm. The following constraints are used by the backtracking algorithm to explore the solution space and construct the paths. In the following notation, the underscore is used in the representation of some tuples to denote "any value", i.e., there is no specific restriction over this value.

1. A tuple in $\mathbb{P}$ has the *Raise* dependency if, and only if, it is the first tuple in $\mathbb{P}$

2. A tuple in $\mathbb{P}$ has the *Handle* dependency if, and only if, it is the last tuple in $\mathbb{P}$

3. A tuple of the form $(x_i, \_, \_)$ is immediately succeeded by a tuple of the form $(x_j, \_, \_)$ in $\mathbb{P}$ if, and only if, the method x_i is immediately succeeded by the method x_j in their call-chain of m

4. A tuple of the form $(x_i, \_, \_)$ is immediately preceded by a tuple of the form $(x_j, \_, \_)$ in $\mathbb{P}$ if, and only if, the method x_i is immediately preceded by the method x_j in their call-chain of m

5. A tuple of the form $(\_, \mathit{Raise}, z)$ is in $\mathbb{P}$ if, and only if, it is immediately succeeded by a tuple of the form: $(\_, \mathit{Handle}, z)$, $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, z)$

6. A tuple of the form $(\_, \mathit{Handle}, z)$ is in $\mathbb{P}$ if, and only if, it is immediately preceded by a tuple of the form: $(\_, \mathit{Raise}, z)$, $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, z)$

7. A tuple with one of the forms $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, (z, \_))$ is in $\mathbb{P}$ if, and only if, it is immediately preceded by a tuple of the form: $(\_, \mathit{Raise}, z)$, $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, z)$

8. A tuple with one of the forms $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, (z, \_))$ is in $\mathbb{P}$ if, and only if, it is immedi-

ately succeeded by a tuple of the form $(\_, \mathit{Handle}, z)$, $(\_, \mathit{Propagate}, z)$, $(\_, \mathit{Re\text{-}throw}, z)$ or $(\_, \mathit{Re\text{-}map}, z)$

Constraints 1 and 2 guarantee that valid paths have an exception raiser and an exception handler in the beginning and in the end of an exception propagation path. Constraints 3 and 4 guarantee that valid paths have its tuples defined for methods in the same order as they relate in the call-chain of m. The other constraints guarantee that every tuple in the exception propagation path is correctly chained from the exception raiser to the exception handler. For example, the Constraint 5 guarantees that if a given method raises an exception of type z, then the next method in the exception propagation path must either handle, propagate, re-throw or re-map this exception. Similarly, Constraint 7 guarantees that in a valid path, if a given method propagates, re-throws or re-maps a given exception of type z, then the previous method in the path must have either raised, propagated, re-thrown or re-mapped an exception of type z. If a given exception propagation path $\mathbb{P}$ satisfies all the constraints listed before, then it is considered a valid path.

The valid exception propagation paths are then used by RAVEN to construct its recommendations. RAVEN constructs its recommendations by comparing valid paths to the call-chain of a method where a violation occurs. To depict how RAVEN constructs its recommendations, consider the following call-chain $\mathbb{C}_{m_i}$ where a violation in localized in a method m_i handling an exception of type t_1 that it is not allowed to:

$$\mathbb{C}_{m_i} = \cdots \quad \underbrace{(m_{i-2})}_{(m_{i-2},\, \mathit{Raise},\, t_1)} \quad \vdash \quad \underbrace{(m_{i-1})}_{(m_{i-1},\, \mathit{Propagate},\, t_1)} \quad \vdash \quad \underbrace{(m_i)}_{(m_i,\, \mathit{Handle},\, t_1)} \quad \cdots$$

In the previous notation, $\alpha \vdash \beta$ means that a method α is immediately succeeded by a method β in a call chain. The text under the braces represents in the form of tuples the exception handling dependencies implemented by each method. The method m_{i-2} raises an exception of type t_1, the method m_{i-1} propagates an exception of type t_1 and the method m_i handles an exception of type t_1.

Now also consider the following exception propagation path $\mathbb{P}_1$ as one of the valid paths produced by RAVEN:

$$\begin{aligned}\mathbb{P}_1 = (m_{i-2}, \mathit{Raise}, t_1) \cdot (m_{i-1}, \mathit{Re\text{-}map}, (t_1, t_2)) \cdot \\ (m_i, \mathit{Propagate}, t_2) \cdot (m_{i+1}, \mathit{Handle}, t_2)\end{aligned}$$

In the path $\mathbb{P}_1$: method m_{i-2} raises an exception of type t_1, method m_{i-1} re-maps an exception of type t_1 to an exception of type t_2, method m_i propagates an exception of type t_2 and method m_{i+1} handles an exception of type t_2. It is worth noticing that in the previous notation used to depict $\mathbb{P}_1$, the third value of the tuple $(m_b, \textit{Re-map}, (t_1, t_2))$ is defined in terms of the pair (t_1, t_2). This notation is only used for the *Re-map* dependency and means that the method m_b re-maps an exception of type t_1 to an exception of the type t_2.

So given the previous call-chain $\mathbb{C}_m$ and the valid exception propagation path $\mathbb{P}_1$, RAVEN produces a recommendation $\mathbb{R}_1$ by comparing the exception handling dependencies implemented by the methods in the call-chain $\mathbb{C}_m$ with the list of tuples in $\mathbb{P}_1$. RAVEN tries to find a set of modifications that if implemented in $\mathbb{C}_m$ would transform its exception propagation path to a path equivalent to $\mathbb{P}_1$. Since $\mathbb{P}_1$ is a valid exception propagation path, RAVEN speculates that by modifying $\mathbb{C}_m$ so that its exception propagation path becomes equivalent to the valid path $\mathbb{P}_1$ it is possible to repair the violation in m. This sequence of modifications serves as a recommendation on how an exception handling violation in a method may be repaired.

RAVEN compares the tuples for the same method. If both tuples are equal, then it considers that it is not necessary to perform any change in the call-chain. If the tuples differ only in the exception type, then RAVEN considers that the exception handling dependency in that method must be changed by only modifying the exception type. If the tuples differ in the exception handling dependency, RAVEN considers that the dependency in the call-chain must be removed and the dependency in the tuple of the exception propagation path must be added.

For the previous example, Table 5.1 depicts the tuples of the call-chain $\mathbb{C}_m$ (left column) and the tuples of the valid path $\mathbb{P}_1$ (right column). Since the method m_{i+1} was not shown on the previous representation of the call-chain, consider that it does not implement any exception handling dependency. In other words, consider that the set of exception handling dependencies extracted from this method is empty (this is represented using the symbol $\varnothing$).

Table 5.1: Comparison Between Tuples

$\mathbb{C}_m$	$\mathbb{P}_1$
$(m_{i-2}, \textit{Raise}, t_1)$	$(m_{i-2}, \textit{Raise}, t_1)$
$(m_{i-1}, \textit{Propagate}, t_1)$	$(m_{i-1}, \textit{Re-map}, (t_1, t_2))$
$(m_i, \textit{Handle}, t_1)$	$(m_i, \textit{Propagate}, t_2)$
$\varnothing$	$(m_{i+1}, \textit{Handle}, t_2)$

By comparing the tuples in $\mathbb{C}_m$ to the tuples in $\mathbb{P}_1$, RAVEN produces the following recommendation $\mathbb{R}_1$:

1. `Remove:` $(m_{i-1}, \textit{Propagate}, t_1)$,
2. `Add:` $(m_{i-1}, \textit{Re-map}, (t_1, t_2))$,
3. `Remove:` $(m_i, \textit{Handle}, t_1)$,
4. `Add:` $(m_i, \textit{Propagate}, t_2)$,
5. `Add:` $(m_{i+1}, \textit{Handle}, t_2)$.

As one can observe in Table 5.1, the first pair of tuples are equal. That is, the first tuple in $\mathbb{C}_m$ is equal to the first tuple in $\mathbb{P}_1$. For this comparison, RAVEN does not recommend any modification. For the second and third pairs of tuples, the tuples differ in the exception handling dependency. By comparing the second pair of tuples, RAVEN recommends that method m_{i-1} should not propagate the exception of type t_1 (Item 1 in $\mathbb{R}_1$); instead, method m_{i-1} should re-map from the type t_1 to the type t_2 (Item 2 in $\mathbb{R}_1$). And by comparing the third pair of tuples, RAVEN recommends that method m_i should not handle the exception of type t_1 (Item 3 in $\mathbb{R}_1$); instead, method m_i should propagate the exception of type t_2 (Item 4 in $\mathbb{R}_1$). By comparing the last pair of tuples, RAVEN recommends that a handler for exceptions of the type t_2 should be added to the method m_{i+1} (Item 5 in $\mathbb{R}_1$). Finally, notice that if we apply these changes to the call-chain $\mathbb{C}_m$, then the violation in m_i is repaired, since it no longer handles the exception of type t_1.

In the previous example, there is no case where the tuples differ only in the exception type. So, consider the following exception propagation path $\mathbb{P}_2$ as another valid path produced by RAVEN:

$$\mathbb{P}_2 = (m_{i-2}, \textit{Raise}, t_2) \cdot (m_{i-1}, \textit{Propagate}, t_2) \cdot (m_i, \textit{Handle}, t_2)$$

In this case, RAVEN performs the following comparisons:

$\mathbb{C}_m$	$\mathbb{P}_2$
$(m_{i-2}, \textit{Raise}, t_1)$	$(m_{i-2}, \textit{Raise}, t_2)$
$(m_{i-1}, \textit{Propagate}, t_1)$	$(m_{i-1}, \textit{Propagate}, t_2)$
$(m_i, \textit{Handle}, t_1)$	$(m_i, \textit{Handle}, t_2)$

By comparing the tuples of the call-chain $\mathbb{C}_m$ and of the valid path $\mathbb{P}_2$, one can observe that only the exception types change. For this reason, RAVEN produces the following recommendation $\mathbb{R}_2$:

1. `Change:` $(m_{i-2}, \mathit{Raise}, t_1) \rightarrow (m_{i-2}, \mathit{Raise}, t_2)$
2. `Change:` $(m_{i-1}, \mathit{Propagate}, t_1) \rightarrow (m_{i-1}, \mathit{Propagate}, t_2)$
3. `Change:` $(m_i, \mathit{Handle}, t_1) \rightarrow (m_{i-1}, \mathit{Handle}, t_2)$

Similarly to the previous recommendation, if we apply these modifications to the call-chain $\mathbb{C}_m$, then the violation in the method m_i is repaired, since it no longer handles the exception of type t_1. Moreover, notice that a difference between exception handling dependencies results in recommendations for two modifications (`Add` and `Remove`), as occurred in the production of the recommendation $\mathbb{R}_1$. On the other hand, differences between exception types results in a recommendation for one modification (`Change`), as occurred in the production of the recommendation $\mathbb{R}_2$. This decision will become clearer in the next section when the ranking scheme of recommendations implemented by RAVEN is discussed.

5.1.3 Ranking Recommendations

The third step in the RAVEN strategy is the ranking of the recommendations constructed during the second step. RAVEN ranks its recommendations by favoring those that require performing fewer modifications. The rationale behind this ranking scheme is that if a recommendation requires fewer modifications to repair a violation, then it is probably easier to implement it than one that requires more changes. Therefore, they should appear in the topmost positions of the list of recommendations, where they can be readily found by developers.

In this context, every list of recommendations $\mathbb{L}$ produced by RAVEN satisfies the following relation:

$$\forall\ \mathbb{R}_i, \mathbb{R}_j \mid \mathbb{R}_i, \mathbb{R}_j \in \mathbb{L} : |\mathbb{R}_i| < |\mathbb{R}_j| \Rightarrow rank(\mathbb{R}_i) < rank(\mathbb{R}_j)$$

where $|\mathbb{R}|$ refers to the number of modifications in a recommendation and $rank(\mathbb{R})$ refers to the position of a recommendation in the list of recommendations. The lower the value of $rank(\mathbb{R})$, the closer $\mathbb{R}$ is to the first positions of the list of recommendations.

For the recommendations presented in the previous section, the following rank is computed:

$$|\mathbb{R}_2| = 3 < |\mathbb{R}_1| = 5 \Rightarrow rank(\mathbb{R}_2) < rank(\mathbb{R}_1)$$

That is, the recommendation $\mathbb{R}_2$ will appear closer to the first positions of the list of recommendations than the recommendation $\mathbb{R}_1$. For these recommendations, in particular, it seems easier to change only the type of the exception, as in the case of the modifications recommended in $\mathbb{R}_2$, than changing exception handling dependencies, as in the case of the modifications recommended in $\mathbb{R}_1$. Moreover, changing one exception handling dependency is considered more difficult than changing only the exception type of an existing dependency. That is why changing an exception handling dependency results in two distinct modifications (`Add` and `Remove`), whereas changing an exception type results in a single modification (`Change`). Thus, the ranking scheme implemented by RAVEN incorporates this assumption about which modifications are more difficult than others.

5.2 Settings of the Evaluation Procedure

This section describes the settings of the procedure conducted to evaluate the RAVEN strategy. In particular, Section 5.2.1 presents the goals of the study, the questions addressed and the metrics employed in the evaluation procedure. Next, Section 5.2.2 details the design of the study and presents its hypothesis. Finally, Section 5.2.3 details the procedure followed to prepare the environment where the evaluation procedure was conducted.

5.2.1 Goals, Questions and Metrics

To structure the methodology of the procedure conducted to evaluate the RAVEN strategy, the Goal-Question-Metric (GQM) method formulated by Basili et al. (BASILI et al., 1994) was used. As previously discussed, RAVEN is the solution that addresses the second research question of this thesis (RQ2). The RAVEN strategy is aimed at providing recommendations of how to repair exception handling violations. In this context, the first goal of this study is stated as follows:

Goal 1: *Evaluate the RAVEN strategy in terms of its effectiveness.*

This study aims at evaluating if the strategy implemented in RAVEN is effective, i.e., whether it is indeed able to produce relevant recommendations. We focused on assessing RAVEN's effectiveness because it positively influences other dimensions related to the adoption of recommendation-based assisting tools (AVAZPOUR et al., 2014). That is, users find no reasons for learning or using tools that are not effective. The RAVEN strategy provides its recommendations as ranked lists. This way, the effectiveness of RAVEN is defined in this study based on a standard quality model for ranking schemes of information retrieval systems (MANNING et al., 2008). In this quality model, effectiveness is characterized in terms of two properties: (i) the ability to produce relevant items and (ii) the ability to rank these items in the topmost positions of the returned list. In the context of RAVEN, relevant recommendations are those able to repair exception handling violations.

In addition, as discussed in Section 5.1, when explicit policy specifications are available, the RAVEN strategy uses them to adjust its solution space. By adjusting its solution space, the RAVEN strategy aims at augmenting the chances of producing relevant recommendations. In this context, the second goal of this evaluation procedure is stated as follows:

> ***Goal 2:*** *Evaluate the RAVEN strategy in terms of the effects of using policy specification in its effectiveness.*

In this context, the following questions are refined from the goals:

> *- Does RAVEN produce relevant recommendations?*
>
> *- Does RAVEN rank relevant recommendations in topmost positions?*
>
> *- Is the effectiveness of RAVEN affected by the use of policy specifications?*

The first two questions are refined from the first goal. These questions are aimed at characterizing the effectiveness of RAVEN according to the adopted quality model. In particular, the first question addresses the first property that characterizes the effectiveness of RAVEN and the second question addresses the second property. The third question is refined form the second goal and it is aimed at investigating the effects of using explicit policy specifications in the effectiveness of RAVEN.

To quantify the effectiveness of RAVEN, a suite of standard metrics used in the evaluation of ranking schemes of information retrieval systems is used. In particular, the Hit, Hit@10 and Reciprocal Rank metrics are employed.

In this evaluation procedure, the Hit metric is computed in the context of a set of exception handling violations. For a given set of exception handling violations, the Hit metric computes the percentage of violations for which a relevant recommendation is produced.

The Hit@10 metric is a variant of the Hit metric. For a given set of exception handling violations, the Hit@10 metric computes the percentage of violations for which a relevant recommendations is ranked within the top 10 positions of the list of recommendations. The Hit@10 metric, in particular, is computed for a list of 10 items due to the default size of items returned by most information retrieval systems (e.g. Google, Bing).

The Reciprocal Rank (RR) metric is computed in the context of a single violation. For a given violation, the Reciprocal Rank metric is computed for the respective list of recommendations $\mathbb{L}$ produced for this violation as follows:

$$RR(\mathbb{L}) = \frac{1}{rank(\mathbb{R})}$$

where $rank(\mathbb{R})$ is the position of the first relevant recommendation $\mathbb{R}$ in the list of recommendations $\mathbb{L}$. If no relevant recommendation is found in the list of recommendations $\mathbb{L}$, then the RR score equals to 0. The Reciprocal Rank metric is quantified based on the actual number of items a user has to inspect before finding a relevant item in the list. Therefore, by using this metric, it is possible to measure an indicator of the underlying effort spent by users to find relevant items.

Finally, the first two questions are addressed based on the analysis of the data collected for the Hit and Hit@10 metrics, respectively. The third question is addressed by analyzing the data collected for the metrics Hit, Hit@10 and Reciprocal Rank metrics.

5.2.2 Study Design and Hypothesis

To achieve the goals and answer the questions defined in the previous section, a laboratory experiment was conducted to employ RAVEN in controlled scenarios of use where exception handling violations needed to be repaired. In particular, a paired comparison experiment (JURISTO and MORENO, 2013), also known as within-subject experiment, was conducted. In this experimental design, the experimental units are exposed to different treatments and each experimental unit serves as its own control. In the context of this evaluation procedure, an experimental unit was the source code of a target system containing an exception handling violation, the independent variable was the use of explicit policy specifications and the dependent variables were the metrics that quantified the effectiveness of RAVEN. Thus, each violation was exposed to two different “treatments” (“RAVEN with policy specifications” and “RAVEN without policy specifications”) and the effectiveness of RAVEN was quantified under each treatment. In addition, the effectiveness of RAVEN under each treatment was compared and the effects of using explicit policy specifications were evaluated by testing the following hypotheses:

Null Hypothesis* H_0*: *The effectiveness of RAVEN is not affected by the use of policy specifications.*
Alternative Hypothesis* H_1*: *The effectiveness of RAVEN is affected by the use of policy specifications.*

In the next section, we present the procedure conducted to prepare the controlled environment in which RAVEN was evaluated.

5.2.3 Preparation Procedure

As discussed in Section 5.2.1, the effectiveness of RAVEN relates to its ability in producing recommendations able to repair exception handling violations. In order to assess if RAVEN produces recommendations that actually repairs violations, first, it was necessary to have the exception handling policy of a program. Unfortunately, most software projects still do not have their exception handling policies explicitly specified. To overcome this limitation, the following controlled environment was set-up to employ RAVEN in scenarios of use where exception handling violations needed to be repaired.

For a given target system, its exception handling policy was inferred from the source code. Then, its source code was adjusted to make it policy-compliant. This version of the target system is called the "policy-compliant version". To specify and verify the exception handling policy of the target systems EPL was used. This language was used because it supports the specification of exception handling rules for all the "canonical" exception handling dependencies typically supported by exception handling mechanisms in programming languages, as described in Chapter 2. Then, violations of the specified policy were injected in the source code of the policy-compliant version. This version of the target system with violations is called the "violating version".

After producing the violating versions, the following procedure was conducted. First, let the exception handling policy produced by analyzing the source code of the target system be called "original policy". Based on the original policy specification, different specifications are produced as follows. From the original policy specification $\mathbb{P}_n$, one of its rules is randomly removed to create the specification $\mathbb{P}_{n-1}$. Next, from the specification $\mathbb{P}_{n-1}$, one of its rules is randomly removed to create the specification $\mathbb{P}_{n-2}$. This is repeated until no rules are left and the empty specification $\mathbb{P}_0$ is produced. This way, the coverage of the policy specification is randomly varied. By employing RAVEN with different policy specifications, different scenarios of use are modeled. These scenarios of use range from a scenario where the development team does not use explicit policy specifications (equivalent to the treatment “RAVEN without policy specification), to a scenario where the development team makes an effort to produce a policy specification that covers a wide spectrum of exception handling design rules (equivalent to the treatment “RAVEN with policy specifications”). The variation of the coverage of the policy specification between these two extreme scenarios of use allows the observation of the effects of different policy specifications in the effectiveness of RAVEN.

Finally, for each violating version and for each different policy specification available, RAVEN was used and the recommendations produced were inspected to check if they were able to repair the violation injected in the respective violating version. If the modifications of a given recommendation were applied to the violating version and these modifications reverted the violating version to the policy-compliant version, then it was considered that the recommendation was able to repair the violation introduced. Thus, the policy-compliant versions were used as the “oracles” for the relevance judgment of the recommendations.

Next, we present the target systems used in the evaluation procedure. We also detail how how the exception handling violations were injected in the target systems.

Target Systems

Target systems were necessary to conduct the evaluation procedure. In particular, target systems with their source code and their intended exception handling policies publicly available. Unfortunately, the vast majority of software projects with publicly available source code do not have their intended exception handling policies available. To overcome this limitation, open-source software projects with stable architectures from which an exception handling policy could be inferred were selected as the target systems.

Thus, three open-source projects were used as target systems: FreeCol, jEdit and Weka. Details of these projects are depicted in Table 5.2. FreeCol is a turn-based strategy game, jEdit is a text editor and Weka is a suite of machine learning tools. All these projects are implemented in Java and are currently hosted in public repositories at Source Forge, where they are amongst the most popular projects in this portal. Moreover, they have been developed for a long period ($>$ 10 years). In addition, these projects are from different domains. Therefore we covered a wide spectrum of how exception handling is typically implemented in real software development environments. In particular, the FreeCol system implements a client-server architecture, so the exception handling responsibilities is divided between the client and

Table 5.2: Target Systems

Project	**Description**	**Version**	**KLOC**
FreeCol	Game	11.5	137
jEdit	Text Editor	5.2.0	109
Weka	Suite of Machine Learning Tools	3.6.0	284

server modules. The jEdit system implements a multi-tier architecture, so its exception handling responsibilities are distributed in each tier. And the Weka system has an architecture with many independent modules and a central module that moderates communication and error handling amongst these modules.

Injection of Exception Handling Violations

To employ RAVEN in the controlled environment described in Section 5.2.2, it was necessary to inject exception handling violations in the source code of the target systems. The violations were manually injected in the source code of the target systems. The effect of violations was isolated by injecting individual violations in the source code of the target systems. For each violation injected, a new version of the source code of the target system was created.

Eventually, the injection of a violation in the source code created compilation errors caused by unreachable *catch* blocks or by uncaught checked exceptions. When compilation errors were caused by unreachable *catch* blocks, these blocks were removed from the source code. When the compilation errors were caused by an uncaught exception, then this exception was propagated from the place where it was created to the nearest enclosing *try* block. If this exception reached a *try* block and no *catch* block captured it, then a new *catch* block declaring the same type of the exception was added to this *try* block. If this exception reached the program entry point (the `main` method), then the exception was propagated and left uncaught. Moreover, if this exception reached a method that could not have its signature changed (e.g., a method overriding a virtual method of an interface), then the exception was handled in this method with a *catch* block declaring the same type of the exception.

In this study, a total of 26 different types of violations were used in the evaluation procedure. These violations were defined based on the study about exception handling faults presented in Chapter 3. In particular, we introduced exception handling violations that simulated faults of commission and faults of omission. We simulated faults of commission by introducing new exception handling dependencies in the source code and also modifying existing ones.

Faults of omission were simulated by removing existing dependencies from the source code. The detailed description of each violation is presented next. The violations are presented in terms of the exception handling dependency that they are related to. They are clustered in Handling Sites Violations, Raising Sites Violations, Re-Mapping Violations and Re-Throwing Violations.

Handling Sites Violations Handling sites violations were introduced by modifying or removing existing handlers and also by introducing new handlers to the source code. A total of 7 different violations in handling sites were used in our evaluation procedure. Next, each one of these violations is detailed.

Added Generic Handler This violation introduces a *try-catch* block to a method through which exceptions traverse. In particular, the *catch* block introduced declares the generic exception type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code. In the following notation, the code snippet on the left represents the source code before the violation is introduced and the code snippet on the right represents the source code after the violation is introduced.

```
                        | void foo(){
void foo() throws T{    |   try{ S }
  S                     |   catch( Exception ) {  }
}                       | }
```

In the previous code snippet, the declaration of the `foo` method encloses a set of statements S. The *try* block is introduced in a manner that it guards S. The *try* block introduced guards all statements of the method declaration. The *catch* block declaring a generic exception type captures all exceptions flowing out of the *try* block. For this reason, the *throws* clause becomes unnecessary and is removed. Moreover, the *catch* block introduced is left empty when the return type of the method is void. When the return type of the method is not void, the *catch* block implements a *return* statement that returns a mock value. In particular, it returns `null` when the return type is an object, `false` when is boolean, `'\u0000'` when is `char` and zero when is a numeric primitive type (`byte`, `short`, `int`, `long`, `float` or `double`).[2]

Added Specific Handler This violation introduces a *try-catch* block to a method through which exceptions traverse. Differently from the violation "Added Generic Handler in Existing Flow", this violation introduces a *catch* block

[2]The character `'\u0000'` is the Unicode representation for the null character.

declaring a specific exception type. The following code snippet exemplifies how this violation is introduced in the source code.

```
                              | void foo() throws T1{
void foo() throws T1, T2{     |   try{ S }
  S                           |   catch( T2 ) { }
}                             | }
```

In the previous code snippet, the *try* block introduced guards all statements of the method declaration. Moreover, the *catch* block captures all exceptions subtypes of T_2 flowing out of the *try* block. This way, the inserted *catch* block interrupts only the propagation of exceptions of a specific type. And for this reason, this type is removed from the *throws* clause. Exceptions of another types continue to flow out of this method. Also, the *catch* block introduced is either empty or it implements the *return* statement the same manner as implemented by the violation “Added Generic Handler”.

Removed Handler This violation removes an existing *catch* block from a method. The exception that was previously captured by the *catch* block is now propagated out of the method. The following code snippet exemplifies how this violation is introduced in the source code.

```
void foo(){                |
  try{ S }                 | void foo() throws T{
  catch( T ) {(...)}       |   S
}                          | }
```

In the previous code snippet, the *catch* block is removed and the respective exception type is declared in the *throws* clause of the method. The exception that was previously captured by the *catch* block flows out of the method.

Changed Handler to Re-Map to Generic Exception This violation introduces a *throw* statement into an existing *catch* block. The exception that was previously captured by the *catch* block is now re-mapped to another type and flows out of the method. In particular, the exception is re-mapped to the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code.

In the previous code snippet, a *throw* statement that instantiates the generic exception type `java.lang.Exception` is introduced in the existing *catch* block. Also, the generic exception type is declared in the *throws* clause

```
void foo(){
  try{ S }
  catch( T ) {
    (...)
  }
}
```

```
void foo() throws Exception{
  try{ S }
  catch( T ) {
    (...)
    throw Exception;
  }
}
```

of the method. The exception caught by the *catch* block is not only re-mapped to the generic type, but it also flows out of the method.

Changed Handler to Re-Map to Unchecked Exception This violation also introduces a *throw* statement into an existing *catch* block. Differently from the violation "Changed Handler to Re-Map to Generic Exception", in this violation, the exception captured by the *catch* block is re-mapped to the unchecked type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code.

```
void foo(){
  try{ S }
  catch( T ) {
    (...)
  }
}
```

```
void foo() {
  try{ S }
  catch( T ) {
    (...)
    throw RuntimeException;
  }
}
```

In the previous code snippets, a *throw* statement that instantiates the unchecked exception type *java.lang.RuntimeException* is introduced in the existing *catch* block. The exception captured by the *catch* block is re-mapped to the unchecked type. Moreover, since unchecked exceptions are automatically propagated by the Java exception handling mechanism, the re-mapped exception is propagated out of the method without need for changing its signature.

Changed Handler to Re-Throw Exception This violation also introduces a *throw* statement into an existing *catch* block. But differently from the previous violations that re-mapped the exception caught by the *catch* block to another type, in this violation, the caught exception is re-thrown. The following code snippet exemplifies how this violation is introduced in the source code.

In the previous code snippet, a *throw* statement that re-throws the caught exception is introduced in the existing *catch* block. In addition, the type of the exception re-thrown is added to the exceptional interface of the method. The exception is not only re-thrown, but it is also propagated out of the method.

```
void foo(){                   | void foo() throws T{
  try{ S }                    |   try{ S }
  catch( T t ) {              |   catch( T t ) {
    (...)                     |     (...)
  }                           |     throw t;
}                             |   }
                              | }
```

Changed Handler to Catch Generic Exception This violation modifies the exception type declared as the argument of an existing *catch* block. In particular, the exception of the existing *catch* block is replaced by the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code.

```
void foo() throws T1{         | void foo() {
  try{ S }                    |   try{ S }
  catch( T2 ) {               |   catch( Exception ) {
    (...)                     |     (...)
  }                           |   }
}                             | }
```

In the previous code snippet, the existing *catch* block is modified by replacing the type of its argument by the generic type `java.lang.Exception`. The generic *catch* block captures exceptions that were previously flowing out of the *try* block. For this reason, the exceptional interface of the method is modified to remove the exceptions that are not propagated out of the method anymore.

Raising Sites Violations Raising sites violations were introduced by modifying or removing existing *throw* statements and also by introducing new *throw* statements to the source code. A total of 6 different types of violations in raising sites were used in our evaluation procedure. Next, each one of these violations is detailed.

Added Generic Raiser This violation adds a new *throw* statement to the source code of a method. In particular, a *throw* statement that raises an exception of the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

In the previous code snippet, the body of the `foo` method is composed of a set of statements S_i. Thus, a new *throw* statement raising an instance of the generic type `java.lang.Exception` is added to the body of the method. In addition, the exceptional interface of the method is modified in order

```
              | void foo() throws Exception {
void foo(){   |   S0
  S0          |   ...
  ...         |   throw Exception;
  Sn          |   ...
}             |   Sn
              | }
```

to propagate to out of the method the exception raised by the new *throw* statement introduced.

Added Unchecked Raiser This violation adds a new *throw* statement to the source code of a method. In particular, it is added a *throw* statement that raises an instance of the unchecked exception type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code:

```
              | void foo() {
void foo(){   |   S0
  S0          |   ...
  ...         |   throw RuntimeException;
  Sn          |   ...
}             |   Sn
              | }
```

In the previous code snippet, a new *throw* statement raising an instance of the unchecked type `java.lang.RuntimeException` is added to the body of an existing method. In addition, the exception raised by the new *throw* statement is not handled within the method; it flows to the callee methods.

Removed Raiser This violation removes an existing *throw* statement from the source code of a method. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E{  |
  S0                  | void foo(){
  ...                 |   S0
  throw E;            |   ...
  ...                 |   Sn
  Sn                  | }
}                     |
```

In the previous code snippet, the existing *throw* statement is removed from the body of an existing method. In addition, the exceptional interface of the modified method is updated by removing the exception that was previously raised and propagated by the method.

Changed Raiser to Raise Generic Exception This violation changes the exception type of an existing *throw* statement. In particular, this violation modifies the exception type of an existing *throw* statement to the generic exception type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T{ | void foo() throws Exception{
  S_0                |   S_0
  ...                |   ...
  throw new T();     |   throw new Exception();
  ...                |   ...
  S_n                |   S_n
}                    | }
```

In the previous code snippet, the existing *throw* statement is modified by changing the type of the exception raised to the generic type `java.lang.Exception`. In addition, the signature of the method is also modified by updating its exceptional interface.

Changed Raiser to Raise Unchecked Exception This violation modifies an existing exceptional flow by changing the exception type of an existing *throw* statement. In particular, this violation modifies the exception type of an existing *throw* statement to the unchecked exception type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E{ | void foo() {
  S_0                |   S_0
  ...                |   ...
  throw new E();     |   throw new RuntimeException();
  ...                |   ...
  S_n                |   S_n
}                    | }
```

In the previous code snippet, the existing *throw* statement is modified by changing the type of the exception raised to the unchecked type `java.lang.RuntimeException`. In addition, the signature of the method is also modified by removing its exceptional interface, since unchecked exceptions are not required to be declared in exceptional interfaces.

Re-Mapping Sites Violations Re-mapping sites violations were introduced by modifying or removing existing *catch* blocks that re-mapped exceptions and also by introducing new *catch* blocks re-mapping exceptions. A total of 8 different types of violations in re-mapping sites were used in our evaluation procedure. Next, each one of these violations is detailed.

Added Re-mapper Raising Generic Exception This violation introduces a new *catch* block that captures an exception and re-maps it to the generic exception type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E{
  S
}
```

```
void foo() throws Exception{
  try{ S }
  catch( E ){
    throw Exception;
  }
}
```

In the previous code snippet, the set of statements S of an existing method `foo` is guarded by a *try* block. A *catch* block is added to capture exceptions subtypes of E. These exceptions were previously propagated by the method. Also, this *catch* block re-maps the caught exception to the generic exception type `java.lang.Exception`. The signature of the method is also modified by declaring the generic exception type in its exceptional interface, so that the re-mapped exception is propagated to out of the method.

Added Re-mapper Raising Unchecked Exception This violation introduces a new *catch* block that captures an exception and re-maps it to the unchecked exception type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E{
  S
}
```

```
void foo(){
  try{ S }
  catch( E ){
    throw RuntimeException;
  }
}
```

In the previous code snippet, the set of statements S of an existing method `foo` is guarded by a *try* block. A *catch* block is added to capture exceptions subtypes of E. These exceptions were previously propagated by the method. Also, this *catch* block re-maps the caught exception to the generic exception type `java.lang.RuntimeException`. The signature of the method is also modified by removing its exceptional interface.

Removed Re-Mapper This violation removes an existing *catch* block that performs a re-mapping. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E2{
  try{ S }
  catch( E1 ){
    throw E2;
  }
}
```

```
void foo() throws E1{
  S
}
```

In the previous code snippet, the *try-catch* block that performs a re-mapping is removed from an existing method `foo`. Also, the exception that was previously captured by the *catch* block is propagated to out of the method. For this reason, the exceptional interface of the method is updated by adding this exception to its exceptional interface.

Changed Re-Mapper to Catch Generic Exception This violation modifies an existing *catch* block that performs a re-mapping by changing the type of the declared argument o the *catch* block to the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T1, T2{
  try{ S }
  catch( E1 ){
    throw T2;
  }
}
```

```
void foo() throws T2{
  try{ S }
  catch( Exception ){
    throw T2;
  }
}
```

In the previous code snippet, by modifying the type of the argument of the *catch* block to the generic type `java.lang.Exception`, this *catch* block captures exceptions subtypes of T_1, which were previously propagated to out of the method. As a consequence, exceptions subtypes of T_1 are also re-mapped to the type T_2. For this reason, exceptions of the type T_1 are not propagated anymore, so the exceptional interface of the method is updated by removing this exception.

Changed Re-Mapper to Re-Map to Generic Exception This violation modifies an existing *catch* block that performs a re-mapping by changing the type of the raised exception to the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T2{  | void foo() throws Exception{
  try{ S }             |   try{ S }
  catch( T1 ){         |   catch( T1 ){
    throw T2;          |     throw Exception;
  }                    |   }
}                      | }
```

In the previous code snippet, the *throw* statement within the *catch* block is modified to raise the generic type `java.lang.Exception`. Moreover, the raised exception is propagated to out of the method. For this reason, the exceptional interface of the method is updated by modifying the previous exception type to the generic exception type.

Changed Re-Mapper to Re-Map to Unchecked Exception This violation modifies an existing *catch* block that performs a re-mapping by changing the type of the raised exception to the unchecked type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T2{  | void foo() {
  try{ S }             |   try{ S }
  catch( T1 ){         |   catch( T1 ){
    throw T2;          |     throw RuntimeException;
  }                    |   }
}                      | }
```

In the previous code snippet, the *throw* statement within the *catch* block is modified to raise the generic type `java.lang.RuntimeException`. Moreover, the exceptional interface of the method is updated to remove the previously propagated exception.

Changed Re-Mapper to Re-throw Exception This violation modifies an existing *catch* block that performs a re-mapping by re-throwing the exception caught by the *catch* block, instead of raising another exception. The following code snippet exemplifies how this violation is introduced in the source code:

In the previous code snippet, the *throw* statement within the *catch* block is modified to raise the exception caught by the *catch* block. Moreover, the

```
void foo() throws T2{  | void foo() throws T1{
  try{ S }             |   try{ S }
  catch( T1 ){         |   catch( T1 t){
    throw T2;          |     throw t;
  }                    |   }
}                      | }
```

exceptional interface of the method is updated to propagate the re-thrown exception.

Changed Re-Mapper to Handle Exception This violation modifies an existing *catch* block that performs a re-mapping by removing the *throw* statement. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T2{  | void foo() {
  try{ S }             |   try{ S }
  catch( T1 ){         |   catch( T1){
    throw T2;          |     (...)
  }                    |   }
}                      | }
```

In the previous code snippet, *catch* block is modified so that it handles the exception, instead of re-mapping it to another type. Moreover, the exceptional interface of the method is updated to remove the exception that was previously propagated.

Re-Throwing Sites Violation Re-throwing sites violations were introduced by modifying or removing existing catch blocks that re-throw exceptions and also by introducing new catch blocks re-throwing exceptions. A total of 6 different types of violations in re-throwing sites were used in our evaluation procedure. Next, each one of these violations is detailed.

Added Re-Thrower This violation introduces a new *catch* block that captures an exception, re-throws it and propagates it to out of the method. The following code snippet exemplifies how this violation is introduced in the source code:

In the previous code snippet, the set of statements S of an existing method `foo` is guarded by a *try* block. Also, a *catch* block captures the exception of type E, re-throws it and propagates it to the out of the method.

```
                       | void foo() throws E{
                       |   try{ S }
void foo() throws E{   |   catch( E e ){
  S                    |     throw e;
}                      |   }
                       | }
```

Removed Re-Thrower This violation removes an existing *catch* block that captures and re-throws an exception to out of the method. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws E{   |
  try{ S }             |
  catch( E e ){        | void foo() throws E{
    (...)              |   S
    throw e;           | }
  }                    |
}                      |
```

In the previous code snippet, the *try-catch* block that guards the set of statements S is removed. No other changes are required.

Changed Re-Thrower to Catch Generic Exception This violation modifies an existing *catch* block that re-throws an exception by changing the type of the argument declared by the *catch* block. In particular, the type of the argument declared by the *catch* block is modified to the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T1{   |
  try{ S }              | void foo() throws Exception {
  catch( T1 t ){        |   try{ S }
    (...)               |   catch( Exception ){
    throw t;            |     throw t;
  }                     |   }
}                       | }
```

In the previous code snippet, the *catch* block is modified so that it captures the generic type `java.lang.Exception`. Moreover, the exceptional interface of the method is updated to propagate the generic exception.

Changed Re-Thrower to Re-Map to Generic Exception This violation modifies an existing *catch* block that re-throws an exception by changing the type of the raised exception to the generic type `java.lang.Exception`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T1{      | void foo() throws Exception {
  try{ S }                 |   try{ S }
  catch( T1 t ){           |   catch( T1 ){
    throw t;               |     throw new Exception();
  }                        |   }
}                          | }
```

In the previous code snippet, the *throw* statement within the *catch* block is modified so that it raises the generic type `java.lang.Exception`. Moreover, the exceptional interface of the method is updated to propagate the generic exception to out of the method.

Changed Re-Thrower to Re-Map to Unchecked Exception This violation modifies an existing *catch* block that re-throws an exception by changing the type of the raised exception to the unchecked type `java.lang.RuntimeException`. The following code snippet exemplifies how this violation is introduced in the source code:

```
void foo() throws T1{      | void foo() {
  try{ S }                 |   try{ S }
  catch( T1 t ){           |   catch( T1 ){
    throw t;               |     throw new RuntimeException();
  }                        |   }
}                          | }
```

In the previous code snippet, the *throw* statement within the *catch* block is modified so that it raises the unchecked type `java.lang.RuntimeException`. Moreover, the exceptional interface of the method is updated to remove the previously propagated exception.

Changed Re-Thrower to Handle Exception This violation modifies an existing *catch* block that re-throws an exception by removing the *throw* statement. The following code snippet exemplifies how this violation is introduced in the source code:

In the previous code snippet, the *throw* statement within the *catch* block is removed. The exception is not re-thrown to out of the block. Moreover, the exceptional interface of the method is updated to remove the previously propagated exception. And a *return* statement is introduced in the *catch* block in the same manner as implemented by the violation "Added Generic Handler".

```
void foo() throws T1{
  try{ S }
  catch( T1 t ){
    throw t;
  }
}
```

```
void foo() {
  try{ S }
  catch( T1 ){
  }
}
```

5.3 Data Analysis

This section presents the analysis of the data collected in the evaluation procedure described in Section 5.2. In particular, Section 5.3.1 provides an overview of the data collected for the metrics defined in Section 5.2.1 and Section 5.3.2 presents the statistical test for the hypothesis presented in Section 5.2.2.

5.3.1 Overview of Collected Data

This section provides an overview of the data collected for the metrics defined in Section 5.2.1. First, the data collected for the Hit and Hit@10 metrics is presented. Then, the data collected for the Reciprocal Rank metric is presented.

Hit and Hit@10 Metrics

The Table 5.3 displays the collected data for the Hit and Hit@10 metrics. Each row in Table 5.3 corresponds to a target system and the last row summarizes the data considering all systems together. The values in the column labelled "N" represent the quantity of violating versions produced for each target system. Moreover, the values of the Hit and Hit@10 metrics are presented in terms of the scenarios in which the policy specification was not used (columns labelled "w/o", i.e.,"without policy") and in which it was used (columns labelled "w/", i.e., "with policy").

Table 5.3: Values of the Hit and Hit@10 Metrics

Target System	**N**	**Hit**		**Hit@10**	
		w/o	*w/*	*w/o*	*w/*
FreeCol	100	0.44	0.97	0.23	0.67
jEdit	98	0.67	1.00	0.44	0.73
Weka	197	0.83	0.96	0.53	0.91
All	395	0.69	0.97	0.43	0.81

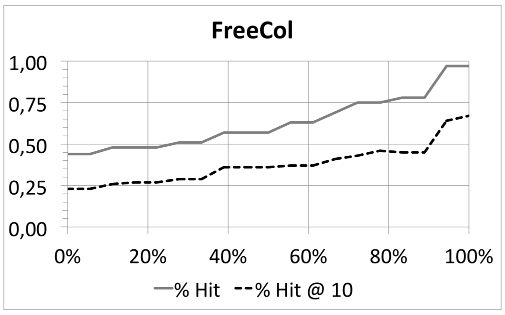
For the scenario in which RAVEN was employed without using explicit policy specifications, the value of the Hit metric was 0.69 and the value of the Hit@10 metric was 0.43, considering all target systems. That is, without using explicit policy specifications, RAVEN produced relevant recommendations for 69% of the violating versions used in the evaluation procedure and ranked relevant recommendations within the top 10 items of the list of recommendations in 43% of the cases. Among the target systems, the highest values for the Hit and Hit@10 metrics were observed in the context of the Weka system: 0.83 and 0.53, respectively. The lowest values for the Hit and Hit@10 metrics were observed in the context of FreeCol: 0.44 and 0.23, respectively.

For the scenario in which explicit policy specifications were available, RAVEN produced relevant recommendations for 97% of the cases and ranked relevant recommendations within the top 10 items of the list of recommendations in 81% of the cases. Among the target systems, the highest values for the Hit metric was observed in the context of jEdit (Hit equals to 1.00), whereas the highest value for the Hit@10 metric was observed in the context of Weka (Hit@10 equals to 0.81). The lowest values for the Hit metric was observed in the context of Weka (Hit equals to 0.96), whereas the lowest value for the Hit@10 metric was observed in the context of FreeCol (Hit@10 equals to 0.67).

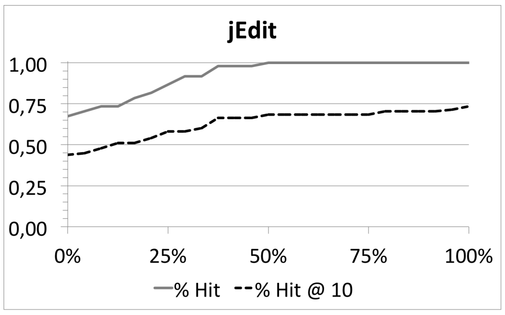
Comparing the values of the metrics observed in each scenario of use of RAVEN, one can observe that for all target systems there is an increase in the values of the metrics Hit and Hit@10 when explicit policy specifications are available. Considering all target systems, there is an increase of 41% for the Hit metric and of 88% for the Hit@10 metric. Among all target systems, the highest increase for the Hit and Hit@10 metrics were observed in the context of the FreeCol system. For this system, we observed an increase of 120% for the Hit metric and of 191% for the Hit@10 metrics. The lowest increase for the Hit metric was observed in the context of Weka (increase of 16%) and the lowest increase for the Hit@10 metric was observed in the context of jEdit (increase of 66%).

The charts depicted in Figure 5.1 present how the values of the metrics Hit and Hit@10 vary according to the different policy specifications used in the context of a each target system. The x-axis in the charts represents the % of rules of the original policy specification used to run RAVEN. The y-axis represents the values of the metrics.

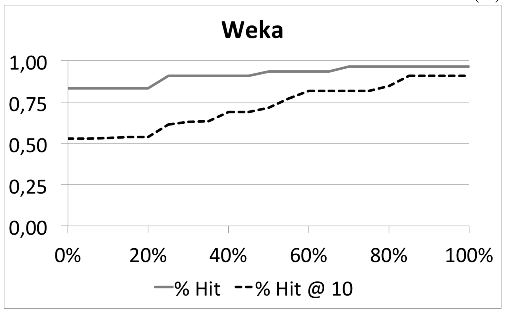
The charts presented in Figure 5.1 depict that as the coverage of the policy specification used to run RAVEN increases, the values of the Hit and Hit@10 metrics tend to increase. There were no cases where an increase in the size of the policy specification was accompanied by a decrease in the values of the Hit and Hit@10 metrics. This tendency was observed in all target systems.



5.1(a): FreeCol



5.1(b): jEdit



5.1(c): Weka

Figure 5.1: Hit and Hit@10 Metrics v.s. Coverage of Policy Specification

For all target systems, the Hit metric reached its maximum value with partial policy specifications. For jEdit, for example, the Hit metric reached the maximum value of 1.0 at the 50% point. This means that it was possible to produce relevant recommendations for all violating versions of jEdit with a policy specification containing 50% of the rules of the original policy. For FreeCol, the Hit metric reached its maximum value of 0.97 at the 95% point. For Weka, the value of the Hit metric reached its maximum value of 0.96 at the 70% point.

The Hit@10 metric reached its maximum value with partial policy specifications for only one target system: Weka. For the Weka target system, the Hit@10 metric reached its maximum value of 0.91 at the 85% point. For FreeCol and jEdit, the values of the Hit@10 metric reached their maximum of 0.67 and 0.73, respectively, at the 100% point.

By observing the the trends of both metrics in each chart, it is possible to observe that after the point where the Hit metric reaches its maximum value and stays constant, the Hit@10 metric continues to increase. For FreeCol, the Hit metric reached its maximum value of 0.97 at the 95% point and after this point the Hit@10 metric increased from 64% to 67%. For jEdit, the Hit metric reached its maximum value of 1.0 at the 50% point and after this point the Hit@10 metric increased from 66% to 73%. For Weka, the Hit metric reached its maximum value of 0.96 at the 70% and after this point the Hit@10 metric

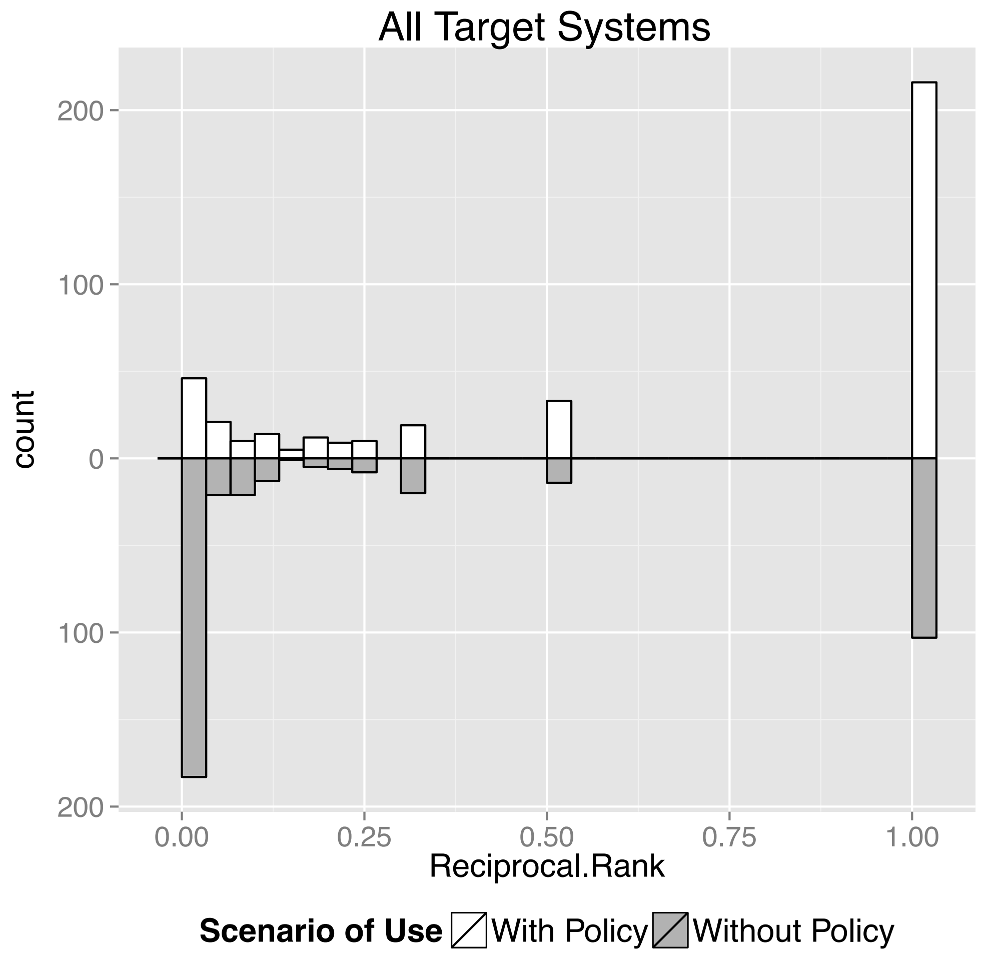


Figure 5.2: Histograms for All Target Systems Together

increased from 82% to 91%.

Reciprocal Rank Metric

The charts displayed in Figures 5.2 and 5.3 present the histograms for the values of the Reciprocal Rank metric. The chart displayed in Figure 5.2 presents the histogram for the data considering all target systems together, whereas the charts displayed in Figure 5.3 presents the histogram for the data in the context of each target system. Each chart contains two histograms, where each histogram presents the values computed in one of the scenarios of use of RAVEN: “With Policy” and “Without Policy”. The histogram for the scenario “With Policy” displays its values above the x-axis, whereas the histogram for the “Without Policy” scenario displays its values below the x-axis.

Moreover, Table 5.4 presents the descriptive statistics for the Reciprocal

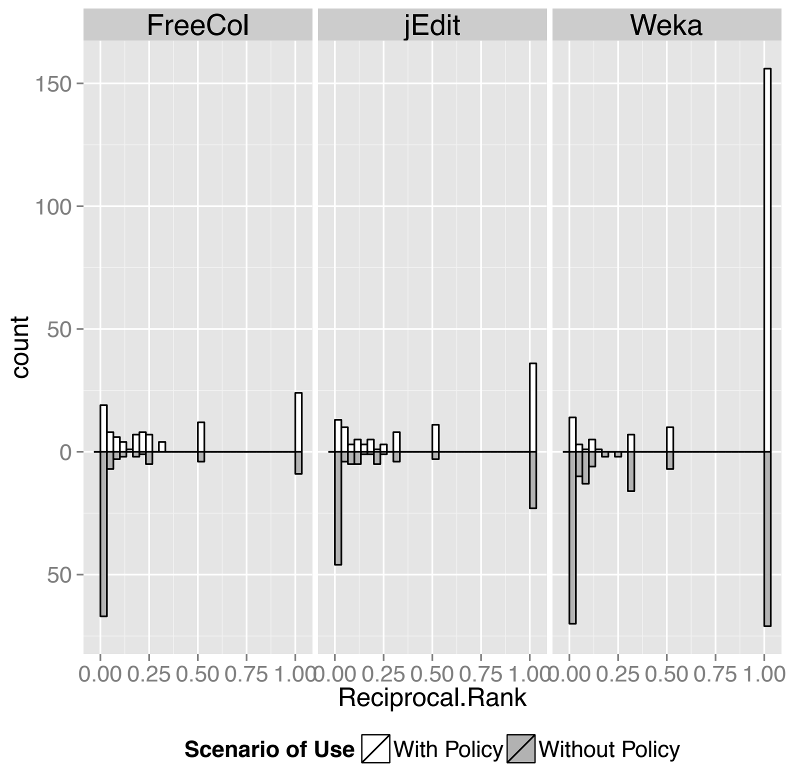


Figure 5.3: Histograms per Individual Target System

Rank scores in each scenario of use: columns labelled “w/o” refer to the scenario of use “Without Policy” and columns labelled “w/” refer to the scenario of use “With Policy”. The row labelled “All” present the descriptive statistics of the Reciprocal Rank scores considering all target systems together, whereas the other rows present the stastistics for each target system. Table 5.5 shows the frequency of the Reciprocal Rank scores in each scenario of use. Each row corresponds to a specific value for the Reciprocal Rank metric (R.R.) and each cell correspond to the number of times a given score occurred.

All Target Systems Together. Consider for now the data collected considering all target systems together. In this case, considering only the scenario of use “Without Policy”, one can observe in the respective histogram shown in Figure 5.2 that most measurements of the Reciprocal Rank metric occurred either around the value 0.0 or around the value 1.0, but mostly around 0.0. In fact, as shown in Table 5.4, the most frequent score – the mode – was 0.0. One can observe in Table 5.5 that there were 121 cases in which the value of the Reciprocal Rank metric was equal to 0.0 (column “All - w/o”, row “R = 0.000”), which is approximately 31% of the total. And there were 103 cases in

Table 5.4: Descriptive Statistics for the Reciprocal Rank Scores

Target Systems	**Mean R.R.**		**Median R.R.**		**Mode R.R.**	
	w/o	*w/*	*w/o*	*w/*	*w/o*	*w/*
FreeCol	0.14	0.38	0.00	0.20	0.00	1.00
jEdit	0.29	0.49	0.06	0.33	0.00	1.00
Weka	0.42	0.83	0.13	1.00	1.00	1.00
All	0.32	0.63	0.05	1.00	0.00	1.00

Table 5.5: Frequency of the Reciprocal Rank Scores

R.R.	**All**		**FreeCol**		**jEdit**		**Weka**	
	w/o	*w/*	*w/o*	*w/*	*w/o*	*w/*	*w/o*	*w/*
R.R. = 0.000	121	10	56	3	32	0	33	7
0.000 <R.R. <0.100	104	67	7	30	23	26	60	11
R.R. = 0.100	5	5	5	0	1	1	2	4
R.R. = 0.111	4	4	4	1	1	2	3	1
R.R. = 0.125	4	5	4	3	3	2	1	0
R.R. = 0.143	1	5	1	1	1	3	0	1
R.R. = 0.167	5	12	2	7	1	5	2	0
R.R. = 0.200	6	9	2	8	5	1	0	0
R.R. = 0.250	8	10	1	7	1	3	2	0
R.R. = 0.333	20	19	5	4	4	8	16	7
R.R. = 0.500	14	33	4	12	3	11	7	10
R.R. = 1.000	103	216	9	24	23	36	71	156

which the value of the Reciprocal Rank metric was equal to 1.0 (column "All - w/o", row "R = 1.000"), which is approximately 26% of the total.

Considering only the scenario of use "With Policy", it is possible to observe in the respective histogram shown in Figure 5.2 that most measurements of the Reciprocal Rank metric occurred around the value 1.0. In fact, as shown in Table 5.4, the mode Reciprocal Rank score was 1.0. One can observe in Table 5.5 that there were 216 cases in which the value of the Reciprocal Rank metric was equal to 1.0 (column "All - w/", row "R = 1.000"), which is approximately 55% of the cases.

Comparing both histograms shown in Figure 5.2, one can observe that they have different shapes. The values in the histogram for the scenario of use "With Policy" are more concentrated to the right, whereas the values in the histogram for the scenario of use "Without Policy" are more concentrated to the left. This tendency can also be observed in Table 5.5. There is a decrease of approximately 92% in the number of occurrences of the values of the Reciprocal Rank equal to 0.0 (column "All", row "R = 0.000") and an increase of approximately 110%in the number of occurrences of the values of this metric equal to 1.0 (column "All", row "R=1.000"). For the mean

Reciprocal Rank, there is an increase of approximately 97%.

Individual Target Systems. Consider for now the data collected in the context of each target system. In this context, considering the "Without Policy" scenario, one can observe in the histograms shown in Figure 5.3 that most values of the Reciprocal Rank metric occurred around the value 0.0 for the FreeCol and jEdit target systems. As shown in Table 5.4, the most frequent score – the mode – for these systems was 0.0. One can observe in Table 5.5 that for FreeCol there were 56 cases in which the value of the Reciprocal Rank metric was equal to 0.0, which accounts for 56% of the total. For jEdit, there were 32 cases equal to 0.0, which is approximately 33% of the total. For the Weka system, most values occur either around the 0.0 value or the 1.0 value, but mostly around 1.0, as can be observed in the respective histogram shown in Figure 5.3. In fact, for Weka the mode Reciprocal Rank score was 1.0. As shown in Table 5.5, there were 71 cases equal to 1.0, which is approximately 36% of the total. And there were 33 cases equal to 0.0, which is approximately 17% of the total.

Considering the "With Policy" scenario, one can observe in the respective histograms shown in Figure 5.3 that for all target systems most values occurred around the value 1.0. In fact, as shown in Table 5.4, the mode score for all target systems was 1.0. As shown in Table 5.5, for FreeCol there were 24 cases in which the value of the Reciprocal Rank metric was equal to 1.0, which accounts for 24% of the total. For jEdit, there were 36 cases equal to 1.0, which is approximately 37% of the total. And for Weka, there were 156 cases equal to 1 .0, which accounts for approximately 79% of the total.

Comparing both scenarios of use, one can observe in the histograms shown in Figure 5.3 a similar trend in all target systems. There is a tendency to decrease the number of values equal to 0.0 accompanied by a tendency to increase the number of values equal to 1.0. As one can observe in Table 5.5, for FreeCol and Weka the number of values equal to 0.0 decreased in approximately 95% and 79%, respectively, whereas for jEdit all values equal to 0.0 vanished away in the scenario "With Policy". Moreover, for FreeCol, jEdit and Weka the number of values equal to 1.0 increased in approximately 166%, 56% and 120%, respectively, and the mean Reciprocal Rank increased in approximately 191%, 66% and 72%, respectively.

5.3.2 Hypothesis Testing

The hypothesis presented in Section 5.2.2 states about the effectiveness of RAVEN. To test the hypothesis, the effectiveness of RAVEN was quantified using the Reciprocal Rank metric. The reasons why this metric was used to quantify the effectiveness of RAVEN in the context of this hypothesis testing were twofold. First, the Reciprocal Rank metric embodies the two properties that characterizes the effectiveness of RAVEN, as defined in Section 5.2.1. That is, lists of recommendations containing relevant recommendations will have a higher Reciprocal Rank score than lists that do not contain relevant recommendations. Also, lists of recommendations that have a relevant recommendation in the topmost positions will have higher Reciprocal Rank scores than those that have a relevant recommendation only in the bottommost positions. Second, the Reciprocal Rank metric is computed in the context of a single violation. This way, each violation used in the evaluation procedure can be exposed to different "treatments" ("RAVEN using policy specifications" and "RAVEN not using policy specifications") and the effectiveness of RAVEN under each "treatment" can be quantified and compared, as designed in Section 5.2.2.

Initially, the collected data for the Reciprocal Rank metric was tested for normality with the Shapiro-Wilk test and for homogeneity of variance with the Levene test. The assumptions about the normality and the homogeneity of variance were rejected. With their rejection, one alternative to meet these assumptions is the transformation of the data. Since the collected data is positive, we transformed the collected data by applying logarithm and squared root transformations, as recommended by Field (FIELD, 2009). The tests for normality and homogeneity of variance were repeated after each transformation. But once again, the assumptions about the normality and the homogeneity of variance were rejected.

Given the rejection of the assumptions about the normality and the homogeneity of variance of the collected data, parametric tests were not adequate to test the hypothesis of this study. For this reason, we employed a non-parametric test for repeated measures. In particular, we employed the Wilcoxon signed rank test.[3] We compared the effectiveness of RAVEN in both "treatments" in the context of each target system, so that the data collected in the context of one target system do not influence the overall result. The results of the Wilcoxon signed rank test are presented in Table 5.6.

[3]All the statistical tests presented in this section were performed using the software IBM SPSS Statistics Desktop version 22.0.0 for Mac OS X

Table 5.6: Result of the Hypothesis Testing

Target System	**N**	**Median (w/o Policy)**	**Median (w/ Policy)**	**Z**	**p**	**r**
FreeCol	100	0,00	0,20	6,792	<0.01	0,68
jEdit	98	0,06	0,33	6,033	<0.01	0,61
Weka	197	0,13	1,00	8,958	<0.01	0,64

The result of the Wilcoxon signed rank test elicited that the effectiveness of RAVEN using policy specifications was significantly different from the effectiveness of RAVEN not using policy specifications for all target systems. Therefore, the effectiveness of RAVEN, as measured by the Reciprocal Rank metric, is indeed affected by the use of policy specifications. In fact, the increase in the medians indicates that the effectiveness of RAVEN is improved when it uses policy specifications. We further discuss this in Section 5.4.

5.4 Results and Findings

This section presents how the data analysis described in Section 5.3 lends support to answer the research questions defined in Section 5.2.1. In particular, Sections 5.4.1, 5.4.2 and 5.4.3 presents the results of the data collected to answer research questions of each one of the three questions of this study, respectively. In addition, during the analysis of the collected data, other findings were also observed. These findings are discussed in Sections 5.4.4 and 5.4.5.

5.4.1 RAVEN Produces Relevant Recommendations

The analysis of the data collected for the Hit metric supports the answer to the first question of this study:

Does RAVEN produce relevant recommendations?

To answer this question, first, consider the scenario of use in which RAVEN was employed without using policy specifications. In this scenario, considering all target systems together, the data presented in the last row of Table 5.3 shows that the value of the Hit metric is 0.69. This means that RAVEN was able to produce relevant recommendations in 69% of all violations analyzed. In the context of the target systems FreeCol, jEdit and Weka the values of the Hit metric were 0.44, 0.67 and 0.83. That is, in the scenario of use "Without Policy", RAVEN produced relevant recommendations in the context of all target systems.

Consider now the scenario of use in which RAVEN was employed using policy specifications. In this scenario, considering all target systems together, the data presented in the last row of Table 5.3 shows that the value of the Hit metric is 0.97. This means that RAVEN was able to produce relevant recommendations in 97% of all violations analyzed when policy specifications were available. In the context of the target systems FreeCol, jEdit and Weka the values of the Hit metric were 0.97, 1.0 and 0.96. That is, in the scenario of use "With Policy", RAVEN produced relevant recommendations in the context of all target systems. In particular, in the context of the jEdit target system, RAVEN was able to produce relevant recommendations for all violations.

In addition, one can observe in the charts in Figure 5.1 that the maximum value of the Hit metric was reached with partial policy specifications in all target systems. For jEdit, for example, the Hit metric reached the maximum value of 1.0 at the 50% point for the jEdit target system. This means that it was possible to produce relevant recommendations for all violating versions of jEdit with a policy specification containing 50% of the rules of the original policy. For FreeCol and Weka, the values of the Hit metric reached their maximum values – 0.97 and 0.96, respectively – at the 95% and 70% point, respectively. This result suggests that RAVEN is able to produce relevant recommendations even with partial policy specifications.

In summary, the RAVEN strategy is indeed able to produce relevant recommendations, even with partial policy specifications. In particular, RAVEN produced relevant recommendations in 69% of the cases when no policy specifications were available and in 97% of the cases when policy specifications were available. The difference between these two scenarios of use of RAVEN suggests that relevant recommendations are more likely to be produced when policy specifications are available than when these specifications are not available. A more in-depth analysis about the effects of using policy specifications in RAVEN's ability to produce relevant recommendations is presented in Section 5.4.3.

5.4.2 RAVEN Ranks Relevant Recommendations in Topmost Positions

The analysis of the data collected for the Hit@10 metric supports the answer to the second research question of this study:

Does RAVEN rank relevant recommendations in topmost positions?

To answer this question, first, consider the scenario of use in which RAVEN was employed without using policy specifications. In this scenario, considering all target systems together, the data presented in the last row of Table 5.3 shows that the value of the Hit@10 metric is 0.43. This means that RAVEN was able to rank a relevant recommendations within the top 10 positions of the list of recommendations in 43% of all violations analyzed. In the context of the target systems FreeCol, jEdit and Weka the values of the Hit metric were 0.23, 0.44 and 0.53. That is, in the scenario of use “Without Policy”, RAVEN ranked relevant recommendations within the top 10 positions of the list of recommendations in the context of all target systems.

Consider now the scenario of use in which RAVEN was employed using policy specifications. In this scenario, considering all target systems together, the data presented in the last row of Table 5.3 shows that the value of the Hit@10 metric is 0.81. This means that RAVEN was able to rank a relevant recommendations within the top 10 positions of the list of recommendations in 83% of all violations analyzed when policy specifications were available. In the context of the target systems FreeCol, jEdit and Weka the values of the Hit metric were 0.67, 0.73 and 0.91. That is, in the scenario of use “With Policy”, RAVEN ranked relevant recommendations within the top 10 positions of the list of recommendations in the context of all target systems.

Differently from the Hit metric, the Hit@10 metric reached its maximum value with partial specifications in only one of the target systems analyzed. For the Weka target system, the maximum value reached by the Hit@10 metric was 0.91 at the 85% point. This means that using a list containing 85% of the rules contained in the original policy, RAVEN ranked relevant recommendations amongst the top 10 positions of the list of recommendations in 91% of the cases. For FreeCol and jEdit, the values of the Hit@10 metric reached their maximum of 0.67 and 0.73 only at the 100% point. One can observe in the charts in Figure 5.1 that after the point where the Hit metric reaches its maximum value, the Hit@10 metric continues to increase its value. This results suggests that the information extracted from policy specifications plays an important role in ranking relevant recommendations in the topmost positions of the list of recommendations.

In summary, the RAVEN strategy is able to rank relevant recommendations within the top 10 positions of the list of recommendations. The RAVEN strategy ranked relevant recommendations within the top 10 positions of the list of recommendations in 43% of the cases when policy specifications were not available and in 83% of the cases when policy specifications were available. This difference observed between the two scenarios of use of RAVEN provides evidence that the use of policy specifications raises the chance of ranking relevant recommendations in the topmost positions of the list of recommendations.

The effects of using policy specifications in RAVEN's ability to rank relevant recommendations in topmost positions of lists of recommendations is presented in Section 5.4.3.

5.4.3 Effectiveness of RAVEN Improves with Policy Specifications

The paired analysis of the collected data in each scenario of use of RAVEN supports the answer to the third research question of this study:

> *Is the effectiveness of RAVEN affected by the use of policy specifications?*

To answer this question, first, consider the data collected for the Hit metric. Comparing the scenario "Without Policy" versus the "With Policy" scenario and considering all target systems, the data presented in the last row of Table 5.3 shows that there is an increase of 41% in the value of the Hit metric when policy specifications are available. In the context of the target systems FreeCol, jEdit and Weka there is an increase of 120%, 49% and 16%, respectively, in the values of the Hit metric when policy specifications are available.

Consider now the data collected for the Hit@10 metric. Comparing the scenario "Without Policy" versus the "With Policy" scenario and considering all target systems, the data presented in the last row of Table 5.3 shows that there is an increase of 88% in the value of the Hit@10 metric when policy specifications are available. In the context of the target systems FreeCol, jEdit and Weka there is an increase of 191%, 66% and 72%, respectively, in the values of the Hit@10 metric when policy specifications are available.

Consider now the data collected for the Reciprocal Rank metric. Comparing the scenario "Without Policy" versus the "With Policy" scenario and considering all target systems, the data presented in the last row of Table 5.4 shows that there is an increase of approximately 97% in the mean Reciprocal Rank score. An increase is also observed for the median Reciprocal Rank and the mode Reciprocal Rank. In fact, the hypothesis testing performed in terms of the Reciprocal Rank metric (Section 5.2.2) elicited a significant difference in the effectiveness of RAVEN when policy specifications are available.

One can observe in the charts in Figure 5.1 some intervals where an increase in the coverage of the policy specification does not affect the values of the Hit and Hit@10 metrics. In some cases, this occurred because the increase in the coverage of the specification moved the position of the first relevant recommendation between two positions out of the top 10 positions. For example, in one of the cases for the FreeCol system, an increase in the coverage of the specification moved a relevant recommendation from the 24th position to the 19th. In this case, no new relevant recommendation was produced, nor a new relevant recommendation reached the top 10. Therefore, the values of the Hit and Hit@10 metrics did not change.

However, there were also cases that an increase in the coverage of the specification did not move the position of the first relevant recommendation, nor produced new relevant recommendations. In these cases, it was observed that the rules added were redundant for the solution space of RAVEN. That is, these rules were used to adjust the solution space by adding information that was already extracted from the source code, or by removing information that was not extracted from the source code. That is, RAVEN tried to add information already present in the solution space, or tried to remove information that was not present in the solution space. Therefore, in these very specific cases, changing the policy specification does not affect the effectiveness of RAVEN.

The results of the paired analysis presented in this section shows that when policy specifications are available, there is an increase in the scores of the metrics that quantify both properties that characterize the effectiveness of RAVEN (see Section 5.2.2). That is, when policy specifications are available, (i) RAVEN's ability to produce relevant recommendations – as measured by the Hit metric – increases in 41% and (ii) RAVEN's ability to rank relevant recommendations in the topmost positions of the lists of recommendations – as measured by the Hit@10 metric – increases in 88%.

Moreover, the effectiveness of RAVEN measured by the Reciprocal Rank metric also increases when policy specifications are available. In particular, the median Reciprocal Rank increased in the context of all target systems. And the hypothesis testing performed shows that the observed difference is statistically significant. Therefore, the results presented in this section provides evidence that the effectiveness of RAVEN is improved by the use of policy specifications.

5.4.4 Mitigating the Cold Start Problem

The results showed that even without using policy specifications, RAVEN produces relevant recommendations. For the Weka system, for example, RAVEN produced relevant recommendations in 83% of the cases without using policy specifications. In this scenario of use, there were sufficient information in the source code to build a solution space that enables the construction of relevant recommendations. However, there might be cases in which there is little information available in the source code. Then, the solution space constructed by RAVEN may be too narrow, which may ultimately hinder the construction of relevant recommendations. For the FreeCol system, for example, RAVEN produced relevant recommendations in 44% of the cases without using policy specifications, which is almost half of the score achieved for Weka. Cases in which there exist little information in the source code may occur when the system is still in its early versions or when a new exception type is used for the first time in the system (e.g., a new third-party library is incorporated *a posteriori* to the system along its evolution). This type of problem caused by insufficient information is commonly referred as the “cold start problem” in recommender systems (JANNACH, 2010).

RAVEN is able to mitigate the cold start problem by complementing the information extracted from the source code with information extracted from policy specifications. Therefore, development teams keen on benefitting from the recommendations of RAVEN in software projects which are still in their early versions, or in cases where exceptions are used for the first time in their projects, should pay more attention to the specification of exception handling policies. In particular, they should focus on the specification of rules that state what exception types their modules are allowed to use because this type of rule is used to add information to the solution space of RAVEN. This way, even when there is few or none information in the source code about how an exception is used, RAVEN will be aware of its intended use. And if the developer uses this exception type in a unintended way, RAVEN will be able to guide him/her in repairing the violation and using the exception type as intended.

5.4.5 Potential Threats in Using Policy Specifications

In this study, we have not observed cases in which an increase in the coverage of the policy specification hindered the effectiveness of RAVEN by either reducing the number of relevant recommendations produced, or by reducing the number of relevant recommendations ranked within the topmost positions of the lists of recommendations. However, we hypothesized the cases in which policy specification may hinder the effectiveness of RAVEN. Thus, software designers and developers can be aware of these cases and consider them while defining their policies.

As discussed in Section 5.1.1, when policy specifications are available, RAVEN uses them to adjust its solution space by pruning and adding tuples.

By adding new tuples to the solution space, RAVEN complements the information extracted from the source code. This way, it has a wider space of possibilities to build relevant recommendations, some of which could not be constructed solely with the information extracted from the source code. But if too many tuples are added to the solution space, this might lead to an overinflated solution space. As a consequence, this might lead to the creation of more irrelevant recommendations. And this effect may move away relevant recommendations from the topmost positions of the returned list.

Similarly, by pruning tuples from the solution space, RAVEN narrows down the information extracted from the source code. Thus, it is possible to discard irrelevant recommendations that could be built with the information extracted from the source code. But if too many tuples are removed from the solution space, this might lead to a solution space so narrow that no relevant recommendation can be built.

Due to the aforementioned reasons, overly detailed policy specifications may impose some threats to the effectiveness of RAVEN. Therefore, policy specifications must be specified focusing on the main exception handling design decisions of a software project. That is, they must focus on the design rules that govern the global exceptions of a program.

Finally, it is worth highlighting that RAVEN takes as input valid policy specifications, but these specifications may be infeasible in the source code. And using infeasible policies may hinder the effectiveness of RAVEN. Consider for example the case of Health Watcher. In Health Watcher, *GUI* and *Business* communicate exclusively through *Façade*. If one policy specification defines that *Business* may only raise `BusinessException` and that only *GUI* is allowed to handle `BusinessException`, it is implicit that these exceptions will have to flow through *Façade*. For this reason, if this same policy specification defines that *Façade* is not allowed to propagate `BusinessException`, then it will not be possible to construct a policy-compliant exception propagation path from *Business* to *GUI* passing through *Façade*. Therefore, in the cases where RAVEN takes as input infeasible specifications, it will not be able to construct recommendations. So far, EPL and RAVEN do not check if policy specifications are feasible in the source code. This is still a developers' responsibility. When RAVEN's algorithm is not able to produce any policy-compliant exception propagation path in a call-chain, this might be a symptom that the specification is not feasible in the source code. We plan to further investigate this hypothesis in future works in order to provide support for checking if policy specifications are feasible in the source code.

5.5 Threats to Validity

This section discusses the study limitations based on the threats to the study validity, presenting the measures taken to mitigate these threats.

5.5.1 Construct Validity

In Section 5.2.2, the effectiveness of RAVEN was characterized based on a quality model typically employed to evaluate ranking schemes of information retrieval systems. This quality mode characterizes effectiveness in terms of the ability to produce relevant items and the ability to rank relevant items in the topmost positions of the returned list. In our evaluation procedure, the Hit and Hit@10 metrics directly quantify these characteristics in the context of sets of exception handling violations. In addition, the Reciprocal Rank metric quantifies if relevant recommendations are produced and also scores higher values when relevant recommendations are in the topmost positions of the returned list. That is, the Reciprocal Rank metric quantifies in the context of a single violation both characteristics used to define the effectiveness of RAVEN. Therefore, we mitigate the threats to the construct validity by employing a standard quality model and suite of metrics to quantify the effectiveness of the proposed recommender heuristic.

5.5.2 Internal Validity

The effectiveness of RAVEN was assessed and compared in different scenarios of use. We mitigated the threats to the internal validity of our study by controlling the settings in which RAVEN was employed and had its effectiveness measured. In the experimental design adopted – the paired comparison experiment – each experimental unit serves as its own control, so the measurements and comparisons were performed between paris of homogeneous experimental units. For this reason, the only difference between the scenarios of use of RAVEN was the variation in the policy specification. This way, we mitigate the threats to the internal validity of our study by minimizing the influence of extraneous variables in the study.

Another threat relates to the relevance judgment used to assess the recommendations produced. We used the policy-compliant versions of the target systems as the “oracles” for the relevance judgment. These versions were produced by the researcher, so they might be a possible bias in the experiment. Without the “oracle”, we would not have one “expected version” to check if the recommendations were properly repairing the violations or not. The relevance of the recommendations would have to be assessed based on opinions of users of RAVEN. Therefore, we are aware of the possible biases in using the “oracle”, but we chose to assess the relevance of the recommendations using consistent and objective criteria, rather than based on subjective opinions of users.

5.5.3 External Validity

The sample of target systems focused on open-source projects. To mitigate this threat, we used as target systems projects from different domains, in order to cover a wide spectrum of how exception handling is typically used in real software development environments. In particular, each target system had a different exception handling design, as discussed in Section 5.2.3.

Another threat to the external validity of this study is the fact that all target systems were implemented in Java. Therefore, the results cannot be extrapolated to projects implemented in other programming languages, specially those with exception handling mechanisms with different characteristics. The automatic reliability checks for checked exceptions is the main difference between the exception handling mechanism implemented by Java and those implemented by other mainstream programming languages. The Java unchecked exceptions, on the other hand, have characteristics similar to those exception handling mechanisms implemented in other mainstream programming languages, like C++ and C#, for example. To mitigate this threat, we employed RAVEN in circumstances similar to those present in other programming languages. In particular, some of the violations introduced in the target systems were related to the use of unchecked exceptions. And no specific differences were observed in the effectiveness of RAVEN for the violating versions where these types of violations were introduced.

5.6 Related Work

As discussed in Section 2.3, current solutions aimed at assisting developers in implementing and maintaining exception handling are focused on support for program comprehension (Section 2.3.1), specification and verification of exception handling properties (Section 2.3.2) and recommendations for implementation of *catch* blocks (Section 2.3.3). There is still no solution aimed at assisting developers in repairing exception handling violations. Therefore, RAVEN is the first solution to assist developers in this task.

In the software architecture literature, there is one research work proposing a solution aimed at assisting developers in repairing architectural violations by means of recommendations. The work of Terra et al. (TERRA et al., 2015) proposes ArchFix, a recommendation engine that provides refactoring guidelines to assist developers in repairing architectural violations. In particular, ArchFix provides a set of 32 pre-defined architectural refactoring recommendations triggered when specific architectural violations are detected in the source code. The architectural violations supported by this solution stem from dependency relations, such as: access, creation, declaration or derivation of specific types; use of specific annotations and propagation of specific exception types. The only types of exception handling violations supported by ArchFix are related to a given method propagating an exception that it is prohibited to propagate, or to a given method not propagating an exception that it is expected to propagate. That is, for the exception handling relations described in Chapter 2, the solution proposed by Terra et al. takes into account only the exception handling dependency relation *Propagate*. Thus, their solution provides only limited support for exception handling violations.

More important than that, the major difference between RAVEN and ArchFix is how the recommendations are produced. The RAVEN strategy leverages on the global context of where exception handling violations occur to construct its recommendations. The ArchFix solution constructs its recommendations by matching pre-defined patterns in the source code. If a given violation is found in the source code and a set of pre-condition is met, then a pre-defined refactoring guideline is recommended. The main limitation of this solution when employed for repairing exception handling violations is its unawareness of the global impact that exceptions might have. Unaware of the global effect of exceptions, the proposed refactoring recommendations may be able to locally repair a given exception handling violation, but the changes performed may actually introduce other violations in the source code. For example, when a method propagates an exception that it is prohibited to, ArchFix recommends to handle the exception in the context of this method. If the repair of this violation requires changing the type of the exception by either changing the type of the exception raised by the *throw* statement, or by re-mapping this exception to another type, then ArchFix is not able to provide recommendations on how to repair the violation. Therefore, when the repair of exception handling violations require global reasoning about the context of the violation, ArchFix is not able to provide recommendations on how to repair them. This is a major limitation, since most exception handling violations are related to global exceptions and, consequently, require global reasoning to repair them.

5.7 Summary

This chapter presented the third contribution of this thesis: RAVEN, a recommender heuristic strategy able to produce recommendations of how to repair exception handling violations. RAVEN produces its recommendations by analyzing the whole call-chain of methods where violations are located. Thus, it considers the global impacts that changes in exception handling code might have in other parts of the program. In addition, RAVEN leverages on information extracted from exception handling policies to improve its effectiveness. The RAVEN strategy supports the repair of exception handling violations in the source code, achieving the second part of the goal of this thesis. Therefore, the cooperation of EPL and RAVEN fulfills our goal of supporting the detection and repair of exception handling violations.

The RAVEN strategy was evaluated in a paired comparison experiment, where it was employed in two different conditions: "Without Policy Specifications" and "With Policy Specifications". The results of this evaluation procedure showed that RAVEN is able to produce relevant recommendations in both conditions. RAVEN is also able to rank relevant recommendations within the topmost positions of the lists of recommendations, even with partial specifications. In fact, the information extracted from the policy specifications seems to play an important role in ranking the relevant recommendations within the topmost positions of the lists of recommendations. The results also showed that the use of policy specifications improved the effectiveness of RAVEN. Finally, RAVEN is the first solution to provide support to the repair of exception handling violations. And the violations in which RAVEN was assessed are similar to the faults observed in the study presented in Chapter 3. Therefore, the results of our evaluation provide initial evidence that RAVEN is a promising solution to support developers in repairing exception handling violations that are potential causes of failures.

6
Conclusion

The development of robust systems requires structuring their internal modules with well-defined exception handling responsibilities. In other words, modules must be structured for detecting runtime errors and for signaling exceptions upon the detection of these errors. Modules must also be structured for detecting exceptions and for implementing actions to respond to these exceptions. In particular, modules should be structured for taking actions for confining the consequences of errors, allowing the system to remain in operation.

Currently, most programming languages provide built-in exception handling mechanisms to support the construction of robust software systems. However, despite being aimed at improving software robustness, the design and implementation of exception handling are often simplified. Worse than that, violations of exception handling policies in the source code are common causes of failures in software systems. To avoid decreases in software robustness caused by failure-causing exception handling violations, developers must be able to detect and repair these violations in the source code. Otherwise, the exception handling code will actually compromise software robustness, instead of improving it.

In this context, the goal of this thesis was to support developers in detecting and repairing exception handling violations in the source code. Two solutions were proposed to fulfill this goal: EPL (Chapter 4) and RAVEN (Chapter 5). EPL is a domain-specific language aimed at assisting the detection of exception handling violations by defining explicit exception handling policies and enforcing them in the source code. RAVEN is a heuristic strategy aimed at supporting the repair of exception handling violations by means of recommendations. EPL and RAVEN are complementary solutions. If policies are documented in EPL, developers have support to detect violations in the source code. Once violations are detected, RAVEN supports their repair. In addition, the RAVEN strategy also benefits from documented policies and leverages on them to improve its effectiveness. Therefore, ,developers would be better-off if they combine them.

6.1 Revisiting the Thesis Contributions

Initially, this thesis aimed at supporting the detection and repair of exception handling faults. For this reason, we conducted an empirical study to investigate what types of exception handling faults occur (Chapter 3). In this study, exception handling faults collected from two open-source projects were investigated. In particular, the fault reports and the modifications performed in the source code to repair the respective fault were analyzed. The exception handling faults were analyzed under two perspectives: the exception handling dependencies implemented and the fault types, i.e., if the faults occurred due to an incorrect dependency (faults of commission), or due to the lack of a dependency (faults of omission).

As the result of the analysis of the exception handling faults, 9 different categories of exception handling faults emerged from the data. In this context, the first contribution of this thesis was:

1st Contribution. *Classification of exception handling faults.*

By better understanding what types of exception handling faults occur, we initially aimed at laying the foundations for investigating proper support for detecting and repairing these faults. However, we observed in this empirical study that most exception handling faults were not related to specific source code patterns, such as empty *catch* blocks or generic *catch* blocks. Therefore, it would be difficult to support their detection by only analyzing the source code structure. We also observed that exception handling faults occurred due to violations of implicit exception handling policies. These results motivated us to move our goal towards the investigation of means to support detecting and repairing violations of exception handling policies.

In this context, we investigated means to support the detection of exception handling violations in the source code. In this direction, we proposed EPL, a domain-specific language for exception handling policies of software systems (Chapter 4). Exception handling policies are expressed in EPL in terms of constraints over the exception handling dependencies that methods in the source code establish with exceptions. In particular, these constraints are expressed in terms of permissions and obligations that software modules have to comply. And violations to permissions and obligations point to potential faults of commission and faults of omission. Therefore, these violations can point to potential sources of failures in the source code.

Both designers and developers can benefit from EPL. Designers have at their disposal a language to explicitly define their intentions regarding the exception handling implementation for their software projects. With an explicit definition about the intended exception handling implementation, developers can readily consult the specification to comprehend how they are supposed to implement the exception handling code. This way, they can prevent the introduction of exception handling violations in the source code. Also, both designers and developers can use the static analyzer to detect exception handling violations in the source code. And by detecting and locating these violations, they can identify parts of the exception handling code that may ultimately lead to subsequent failures.

In this context, the second contribution of this thesis was:

2nd Contribution. *Domain-specific language for specifying and verifying exception handling policies that supports the detection of exception handling violations.*

With EPL, we achieved the first part of the goal of this thesis, which is supporting the detection of exception handling violations. The remaining part of the goal of this thesis was supporting the repair of these violations. In this direction, we proposed RAVEN, a recommender heuristic strategy for assisting the repair of exception handling violations (Chapter 5). The RAVEN strategy provides recommendations of how a given exception handling violation can be repaired. To do so, the heuristic is aware of the global impact of exceptions by taking into account the whole context of where a given exception handling violation is localized. This way, these recommendations serve as detailed blueprints of how developers should modify the source code in order to repair exception handling violations. In addition, the violations that RAVEN is able to repair are similar to the faults observed in the study presented in Chapter 3. Therefore, RAVEN is a promising solution to support developers in repairing exception handling violations that are potential causes of failures.

In this context, the third contribution of this thesis was:

3^{rd} ***Contribution.*** *Recommender heuristic strategy that supports the repair of exception handling violations.*

Finally, to facilitate future references to works that resulted from this thesis, Table 6.1 presents the papers produced in the context of this thesis and the respective chapters to which they relate.

Table 6.1: Papers Produced in the Context of this Thesis

Paper	Chapter
E. A. Barbosa and A. Garcia. *Categorizing Faults in Exception Handling: A Study of Open Source Projects.* In Proceedings of the 28th Brazilian Symposium on Software Engineering (SBES'14), 2014.	3
E. A. Barbosa. *Improving exception handling with recommendations.* In Companion Proceedings of the 36th International Conference on Software Engineering - Doctoral Symposium (ICSE'14), 2014.	4, 5
E. A. Barbosa. *Mastering Global Exceptions with Policy-Aware Recommendations.* In Companion Proceedings of the 37th International Conference on Software Engineering - ACM Research Competition (ICSE'15), 2015.	4, 5
E. A. Barbosa, A. Garcia, M. Robillard and B. Jakobus. *Enforcing exception handling policies with a domain-specific language.* In IEEE Transactions on Software Engineering, Dec., 2015.	4
N. Cacho, **E. A. Barbosa**, T. Cesar, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia and A. Garcia. *Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs.* In Proceedings of the 36th International Conference on Software Engineering (ICSE'14), 2014.	2
N. Cacho, **E. A. Barbosa**, J. Araujo, F. Pranto, A. Garcia, T. Cesar, A. Cassio, E. Soares, T. Filipe and I. Garcia. *How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications.* In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'14), 2014.	2
B. Jakobus, A. Garcia, **E. A. Barbosa** and C. J. Lucena. *Contrasting exception handling code across languages: An analysis of 50 open source projects.* In 26th International Symposium on Software Reliability Engineering (ISSRE'15), 2015.	2

6.2 Future Work

Along the studies conducted in the context of this thesis, new challenges have emerged. Next, further directions for future works are presented.

Further Investigation in Other Programming Languages. The studies conducted in the context of this thesis were focused on the exception handling mechanisms of the Java programming language. In fact, most empirical studies conducted to analyze exception handling in software systems were conducted in the context of Java programs. Recently, we conducted empirical studies in collaboration with Cacho et al. and with Jakobus et al. to further investigate exception handling implementation in other programming languages. In particular, the study conducted in collaboration with Cacho et al. investigated the impact of exception handling in software robustness in Java and C# programs. And the study conducted with Jakobus et al. assessed the use of exception handling mechanisms in Java, C#, C++, PHP and JavaScript programs. Even so, studies still need to be conducted to further investigate exception handling in other programming languages.

In addition, no work has analyzed exception handling in the context of multi-language systems. That is, there is still no empirical knowledge about how exception handling is designed and implemented in the context of software systems with modules written in different programming languages. It seems interesting to investigate how the responsibilities of exception handling are designed and implemented among the modules written in different programming languages.

As a starting point for these future works, the studies conducted in the context of this thesis could be replicated to analyze multi-language systems. This way, it could be investigated what kinds of exception handling faults occur when the exception handling responsibilities are divided among modules written in different programming languages. Also, how exception handling policies can be defined and enforced in multi-language systems. And how to support the detection and repair of exception handling violations in multi-language systems.

Recover Exception Handling Policy from the Source Code. It would be interesting to provide support for recovering an exception handling policy from the exception handling code already implemented in the system. This would be useful in scenarios where the system is already in production, but without an exception handling policy. This way, the development team can start the definition of the exception handling policy by first recovering a policy from the source code. And then, they can refine this policy to reflect their intentions regarding the exception handling implementation.

One first difficulty in recovering a policy expressed as obligations and permissions from the source code is how to infer the semantics of constraints over the exception handling dependencies directly from the source code. For example, given one method that raises a given exception, it is not straightforward to define if this should be expressed as an obligation or a permission. One possible support for recovering an exception handling policy from the source code could be defined as follows. First, the developer defines the compartments of his policy. Then, a supporting tool extracts the exception handling dependencies and the respective exception types that each compartment establishes in the source code. The supporting tool would only present to developer a summary of the implemented exception handling dependencies in each compartment. The developer would be responsible for defining the semantics of the intended constraints over each exception handling dependency. This way, this supporting tool would only assist the developer in recovering the exception handling policy from the source code. It would not be able to actually recover it.

Another possible support for recovering an exception handling policy from the source code is to try to infer the semantics of the constraints by analyzing the source code evolution history. This possible support could be defined as follows. First, the developer defines the compartments of his policy. Then, the supporting tool would analyze the version history of the system to recover the rules of the policy. If a given exception handling dependency exists in the source code and the exception type used was modified in a previous version, then this might indicate that the compartment is not allowed to relate to the previous exception type. It might also indicate that the compartment is allowed, or obligated, to relate to the current exception type. Similarly, if a given exception handling dependency is removed from a method along software evolution and the respective compartment does not establish this dependency anymore, then this might indicate that the respective compartment is not allowed to establish this dependency. This way, these heuristic strategies could be further developed and tested in future works to assess if they are able to automatically recover exception handling policies from the version history of a system.

Improved Tool Support for the Recommendations. The recommendations produced by the RAVEN strategy are sequences of modifications that should be performed in the source code to repair exception handling violations. One first improvement necessary to make RAVEN usable in practice is to incorporate it to mainstream IDEs, such as Eclipse and NetBeans. This way, it is expected that developers incorporate RAVEN to their development activities more easily than if RAVEN was provided as a stand-alone application independent of their IDEs. We believe that incorporating RAVEN to Eclipse can be achieved in a short-term future work, since it was already implemented as an Eclipse plugin. So to actually incorporate RAVEN to Eclipse is necessary to create a graphical user interface and integrate it to the RAVEN plugin.

It would also be interesting to combine the recommendations provided by RAVEN with tools for automating modifications in the source code. These tools would be similar to automated refactoring tools provided in mainstream IDEs, such as Eclipse and NetBeans. Thus, developers could choose one recommendation and part of the recommended modifications in the source code would be automated. One major challenge in automating modifications in the exception handling code would be to define the proper level to which *try* blocks would be applied. In most mainstream programming languages, like Java, C#, JavaScript, and the like, *try* blocks are attached to blocks of statements. This means that *try* blocks may guard only one statement, but also all statements of a method. Therefore, there are many different ways in which *try* blocks can be added to source code and these differences may have impact in the program behavior. The proper way of automatically modifying exception handling code without introducing any negative side-effect in the program is still to be investigated.

Even More Global Recommender Heuristics. The recommendations provided by RAVEN are produced by taking into account the global context of exception handling violations. More specifically, RAVEN analyzes the whole call-chain of the method where a violation occurs. One possible evolution for RAVEN would be investigating means to provide support for repairing all exception handling violations in the context of a compartment, or even the whole system, for example. A starting point for this alternative could be investigating whether for different exception handling violations occurring in the same compartment the recommendations provided by RAVEN conflict with each other. In other words, if one repairing modification recommended for one violation conflicts with the modifications recommended for other violations. This way, an improved version of RAVEN could favor recommendations that do not conflict with repairing modifications of other violations. Thus, future works may further investigate even more global recommender heuristic strategies for repairing multiple exception handling violations in the source code at once.

Bibliography

ABRANTES and COELHO, 2015. J. Abrantes and R. Coelho. **Specifying and dynamically monitoring the exception handling policy**. In *Proc. of the 27 International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2015. 1.1.1, 1.1.2, 2.3.2, 4.5, 4.5

ARCOVERDE et al., 2013. R. Arcoverde, E. Guimarães, I. Macia, A. Garcia and Y. Cai. **Prioritization of code anomalies based on architecture sensitiveness**. In *Software Engineering (SBES), 2013 27th Brazilian Symposium on*, pages 69–78. IEEE, 2013. 4.3.1, 4.4.1, 4.4.1

AVAZPOUR et al., 2014. I. Avazpour, T. Pitakrat, L. Grunske and J. Grundy. **Dimensions and metrics for evaluating recommendation systems**. In *Recommendation Systems in Software Engineering*, pages 245–273. Springer Berlin Heidelberg, 2014. 5.2.1

AVIZIENIS et al., 2004. A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr. **Basic concepts and taxonomy of dependable and secure computing**. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004. 1, 2.1

BACHMANN and BERNSTEIN, 2009. A. Bachmann and A. Bernstein. **Data retrieval, processing and linking for software process data analysis**. Technical report, University of Zurich, 2009. 3.1.2

BARBOSA and GARCIA, 2011. E. A. Barbosa and A. Garcia. **Analyzing Exceptional Interfaces on Evolving Frameworks**. In *2011 Fifth Latin-American Symposium on Dependable Computing Workshops*, pages 17–20. IEEE, Apr. 2011. 4.2.1

BARBOSA et al., 2012. E. A. Barbosa, A. Garcia and M. Mezini. **A recommendation system for exception handling code**. In *Proceedings of the 5th International Workshop on Exception Handling (WEH)*, pages 52–54. IEEE, June 2012. 1.1.2, 2.3.3

BARBOSA et al., 2012a. E. A. Barbosa, A. Garcia and M. Mezini. **Heuristic Strategies for Recommendation of Exception Handling Code**. In *Proceedings of the 2012 26th Brazilian Symposium on Software Engineering SBES '12*, pages 171–180, Washington, DC, USA, Sept. 2012. IEEE. 1.1.2, 2.3.3

BARBOSA et al., 2014. E. A. Barbosa, A. Garcia and S. D. J. Barbosa. **Categorizing Faults in Exception Handling: A Study of Open Source Projects**. In *Proceedings of the XXVIII Brazilian Symposium on Software Engineering (SBES'14)*, 2014. 3.6

BARBOSA et al., 2015. E. A. Barbosa, A. Garcia, M. Robillard and B. Jakobus. **Enforcing exception handling policies with a domain-specific language**. *Accepted to appear in IEEE Transactions on Software Engineering*, 2015. 4.6

BASILI et al., 1994. V. Basili, G. Caldiera and D. H. Rombach. **The goal question metric approach**. *Encyclopedia of Software Engineering*, 1994. 5.2.1

BLOCH, 2008. J. Bloch. **Effective java**. *The Java Series*. Prentice Hall, 2008. 2.1.2

BRITO et al., 2009. P. H. S. Brito, R. Lemos, C. M. F. Rubira and E. Martins. **Architecting Fault Tolerance with Exception Handling: Verification and Validation**. *Journal of Computer Science and Technology*, 24(2):212–237, Apr. 2009. 4.3.1

BROOKS, 1983. R. Brooks. **Towards a theory of the comprehension of computer programs**. *International journal of man-machine studies*, 18(6):543–554, 1983. 2.3.1

BRUTNIK et al., 2006. M. Bruntink, A. Van Deursen and T. Tourwé. **Discovering faults in idiom-based exception handling**. In *Proceedings of the 28th international conference on Software engineering*, pages 242–251. ACM, 2006. 1.2, 3, 3.3.2, 3.5, 3.6

BUHR, 2000. P. A. Buhr and W. Y. R. Mok. **Advanced Exception Handling Mechanisms**. *IEEE Transactions on Software Engineering*, 26(9):820, 2000. 1, 2.1.1

BURROWS et al., 2010. R. Burrows, F. C. Ferrari, O. A. Lemos, A. Garcia and F. Taiani. **The impact of coupling on the fault-proneness of**

aspect-oriented programs: an empirical study. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 329–338. IEEE, 2010. 4.4.1, 4.4.2, 4.4.2

BUSE and WEIMER, 2008. R. P. Buse and W. R. Weimer. **Automatic documentation inference for exceptions**. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 273–282. ACM, 2008. 1.1.1, 4

CABRAL and MARQUES, 2007. B. Cabral and P. Marques. **Exception handling: A field study in java and .net**, volume 4609 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 2.2.1

CABRAL and MARQUES, 2008. B. Cabral and P. Marques. **A case for automatic exception handling**. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 403–406. IEEE Computer Society, 2008. 2.3.3

CACHO et al., 2008. N. Cacho, F. Castor, A. Garcia and E. Figueiredo. **EJFlow : Taming Exceptional Control Flows in Aspect-Oriented Programming**. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 72–83, 2008. 1, 1.1.1, 1.1.2, 2.3.2, 4.3.1, 4.4.1, 4.4.1, 4.5

CACHO et al., 2009. N. Cacho, F. Dantas, A. Garcia and F. Castor. **Exception Flows Made Explicit: An Exploratory Study**. In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, pages 43–53, 2009. 1, 1.1.2

CACHO et al., 2014a. N. Cacho, E. A. Barbosa, J. Araújo, F. Pranto, A. Garcia, T. César, A. Cassio, E. Soares, T. Filipe and I. Garcia. **How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications**. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014. 1, 1.1, 1.1, 1.1.1, 1.1.2, 2.2.2, 2.3.2, 2.3.3, 2.4, 3.5, 4.2.1

CACHO et al., 2014b. N. Cacho, E. A. Barbosa, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia and A. Garcia. **Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs**. In *Proceedings of the 36th International Conference on Software Engineering*, pages 584–595, New York, New York, USA, 2014. ACM. 1, 1.1, 1.1, 1.1.1, 1.1.2, 2.2.2, 2.3.2, 2.3.3, 2.4, 3.5

CHANG et al., 2001. B.-M. Chang, J.-W. Jo, K. Yi and K.-M. Choe. **Interprocedural exception analysis for Java**. In *Proceedings of the 2001 ACM symposium on Applied computing - SAC '01*, pages 620–625, New York, New York, USA, Mar. 2001. ACM Press. 1.1.2, 2.3.1

CHELLAS, 1980. B. F. Chellas. **Modal logic: an introduction**, volume 316. Cambridge Univ Press, 1980. 4

CLEMENTS, 1996. P. C. Clements. **A Survey of Architecture Description Languages**. In *Proceedings of the 8th International Workshop on Software Specification and Design*, page 16. IEEE Computer Society, Mar. 1996. 4.5

COELHO et al., 2008. R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa and C. Lucena. **Assessing the Impact of Aspects on Exception Flows: An Exploratory Study**, volume 5142 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, July 2008. 1, 1.1, 1.1.1, 1.1.2, 2.2.2, 3.5, 4.3.1, 4.4.1, 4.4.1

CRISTIAN, 1989. F. Cristian. **Exception Handling**. In *Dependability of Resilient Computers*, pages 68–97, 1989. 1, 2.1.1

DAGENAIS and ROBILLARD, 2011. B. Dagenais and M. P. Robillard. **Recommending Adaptive Changes for Framework Evolution**. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–35, Sept. 2011. 3.1.2

DAVIS, 1989. F. D. Davis. **Perceived usefulness, perceived ease of use, and user acceptance of information technology**. *MIS quarterly*, pages 319–340, 1989. 4.3.1, 4.3.3

DELEMOS and ROMANOVSKY, 2001. R. de Lemos and A. Romanovsky. **Exception Handling in the Software Lifecycle**. *International Journal of Computer Systems Science and Engineering*, Mar. 2001. 1.1.1, 4, 4.3.1

EBERT and CASTOR, 2013. F. Ebert and F. Castor. **A Study on Developers' Perceptions about Exception Handling Bugs**. In *2013 IEEE International Conference on Software Maintenance*, pages 448–451. IEEE, Sept. 2013. 1.1.1, 4

EBERT et al., 2015. F. Ebert, F. Castor and A. Serebrenik. **An exploratory study on exception handling bugs in java programs**. *Journal of Systems and Software*, 106:82–101, 2015. 1.1.1, 1.1.2, 3.5, 4

EICHBERG et al., 2008. M. Eichberg, S. Kloppenburg, K. Klose and M. Mezini. **Defining and continuous checking of structural program dependencies**. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 391, New York, New York, USA, May 2008. ACM Press. 4.5, 4.5

FERRARI et al., 2010. F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares et al. **An exploratory study of fault-proneness in evolving aspect-oriented programs**. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 65–74. ACM, 2010. 4.4.1, 4.4.2, 4.4.2

FIELD, 2009. A. Field. **Discovering statistics using spss**. Sage publications, 2009. 5.3.2

FU et al., 2005. C. Fu, A. Milanova, B. G. Ryder and D. G. Wonnacott. **Robustness testing of java server applications**. *Software Engineering, IEEE Transactions on*, 31(4):292–311, 2005. 1.1.1, 1.1.2, 3.3.1, 4

FU and RYDER, 2007. C. Fu and B. 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. IEEE, May 2007. 1.1.1, 1.1.2, 2.3.1, 2.3.2

GARCIA et al., 2001. A. F. Garcia, C. M. Rubira, A. Romanovsky and J. Xu. **A comparative study of exception handling mechanisms for building dependable object-oriented software**. *Journal of Systems and Software*, 59(2):197–222, Nov. 2001. 1, 2.1.1

GARCIA et al., 2013. J. Garcia, I. Ivkovic and N. Medvidovic. **A comparative analysis of software architecture recovery techniques**. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496. IEEE, 2013. 4.3.2

GOODENOUGH, 1975. J. B. Goodenough. **Exception handling: issues and a proposed notation**. *Communications of the ACM*, 18(12):683, 1975. 1, 2.1.1

GURGEL et al., 2014. A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg and R. Mitschke. **Blending and reusing rules for architectural degradation prevention**. In *Proceedings of the 13th international* *conference on Modularity - MODULARITY '14*, pages 61–72, New York, New York, USA, Apr. 2014. ACM Press. 4.5, 4.5

HIGO and KUSUMOTO, 2014. Y. Higo and S. Kusumoto. **How Should We Measure Functional Sameness from Program Source Code? – An Exploratory Study on Java Methods**. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. ACM Press, 2014. 5.1.1

HOLMES and MURPHY, 2005. R. Holmes and G. C. Murphy. **Using structural context to recommend source code examples**. In *Proceedings of the 27th international conference on Software engineering*, pages 117–125. ACM, 2005. 5.1.1

IEEE, 1990. IEEE Computer Society. **IEEE Standard Glossary of Software Engineering Terminology**, Dec. 1990. 1, 2.1

JSL-6. Oracle. **The java language specification: Java se 6 edition**. https://docs.oracle.com/javase/specs/jls/se6/html/j3TOC.html, September 2015. 2.1.2, 2.1.2, 2.1.2, 2.1.2, 2.1.2, 2.1.2, 4.2.1, 4.2.1

JSR-342. JCP. **Jsr 342: Java platform enterprise edition 7 specification**. https://jcp.org/en/jsr/detail?id=342. 4.4.1, 4.4.2, 4.4.2

JAKOBUS et al., 2015. B. Jakobus, A. Garcia, E. A. Barbosa and C. J. Lucena. **Contrasting exception handling code across languages: An analysis of 50 open source projects**. In *26th International Symposium on Software Reliability Engineering (ISSRE 2015)*, 2015. 1, 2.1.1, 2.2.1, 2.4

JANNACH, 2010. D. Jannach, M. Zanker, A. Felfernig and G. Friedrich. **Recommender systems: an introduction**. Cambridge University Press, 2010. 5.4.4

JURISTO and MORENO, 2013. N. Juristo and A. M. Moreno. **Basics of software engineering experimentation**. Springer Science & Business Media, 2013. 5.2.2

KIENZLE, 2008. J. Kienzle. **On exceptions and the software development life cycle**. In *Proceedings of the 4th international workshop on Exception handling - WEH '08*, pages 32–38, New York, New York, USA, Nov. 2008. ACM Press. 1.1.1, 4, 4.3.1

KNODEL and POPESCU, 2007. J. Knodel and D. Popescu. **A Comparison of Static Architecture Compliance Checking Approaches**. In *2007*

Working IEEE/IFIP Conference on Software Architecture (WICSA'07), pages 12–12. IEEE, Jan. 2007. 4.5

KULESZA et al. 2006. U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. Staa and C. Lucena. **Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study**. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 223–233. IEEE, Sept. 2006. 1.1

LEE and ANDERSON, 1990. P. A. Lee and T. Anderson. **Fault tolerance: Principles and practice**. Springer-Verlag, 1990. 1, 2.1, 2.1.1, 2.1.1

LITKE, 1999. J. D. Litke. **A systematic approach for implementing fault tolerant software designs in Ada**. In *Proceedings of the conference on TRI-ADA '90 - TRI-Ada '90*, pages 403–408, New York, New York, USA, Dec. 1990. ACM Press. 4.5

MACIA et al., 2012. I. Macia, R. Arcoverde, A. Garcia, C. Chavez and A. von Staa. **On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms**. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 277–286. IEEE, Mar. 2012. 4.3.1, 4.4.1, 4.4.1

MACIA et al., 2012a. I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic and A. von Staa. **Are automatically-detected code anomalies relevant to architectural modularity?** In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*, page 167, New York, New York, USA, Mar. 2012. ACM Press. 4.3.1, 4.4.1, 4.4.1

MALAYERI and ALDRICH, 2006. D. Malayeri and J. Aldrich. **Practical Exception Specifications**. In *Advanced Topics in Exception Handling Techniques*, pages 200–220. Springer, 2006. 4.5

MANNING et al., 2008. C. D. Manning, P. Raghavan, H. Schütze et al. **Introduction to information retrieval**, volume 1. Cambridge university press Cambridge, 2008. 5.2.1

MARINESCU, 2011. C. Marinescu. **Are the classes that use exceptions defect prone?** In *Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11*, page 56, New York, New York, USA, Sept. 2011. ACM Press. 1, 2.2.2

MARINESCU, 2013. C. Marinescu. **Should We Beware the Exceptions? An Empirical Study on the Eclipse Project**. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 250–257. IEEE, Sept. 2013. 1, 2.2.2

RAHMAN and ROY, 2014. M. M. Rahman and C. K. Roy. **On the use of context in recommending exception handling code examples**. In *14th International Working Conference on Source Code Analysis and Manipulation*, pages 285–294. IEEE Computer Society, 2014. 1.1.2, 2.3.3

ROBILLARD and MURPHY, 1999. M. P. Robillard and G. C. Murphy. **Analyzing Exception Flow in Java Programs**, 1999. 1, 1.1.2, 2.3.1

ROBILLARD and MURPHY, 2000. M. P. Robillard and G. C. Murphy. **Designing robust Java programs with exceptions**. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, pages 2 – 10, 2000. 4.5

ROBILLARD and MURPHY, 2003. M. P. Robillard and G. C. Murphy. **Static analysis to support the evolution of exception structure in object-oriented systems**. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003. 1, 1.1.2, 2.3.1

ROBILLARD et al., 2010. M. Robillard, R. Walker and T. Zimmermann. **Recommendation Systems for Software Engineering**. *IEEE Software*, 27(4):80–86, July 2010. 2.3.3

SAHAVECHAPHAN and CLAYPOOL, 2006. N. Sahavechaphan and K. Claypool. **Xsnippet: mining for sample code**. *ACM Sigplan Notices*, 41(10):413–430, 2006. 5.1.1

SALES and COELHO, 2011. R. Sales, RJ; Coelho. **Preserving the exception handling design rules in software product line context: A practical approach**. In *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*, pages 9–16. IEEE, 2011. 1.1.1, 1.1.2, 2.3.2, 4.3.1, 4.4.1, 4.4.1, 4.5, 4.5

SAWADPONG et al., 2012. P. Sawadpong, E. B. Allen and B. J. Williams. **Exception Handling Defects: An Empirical Study**. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, pages 90–97. IEEE, Oct. 2012. 1, 2.2.2

SHAH, GÖRG and HARROLD, 2008a. H. Shah, C. Görg and M. J. Harrold. **Visualization of exception handling constructs to support program**

understanding. In *Proceedings of the 4th ACM Symposium on Software Visualization*, page 19, New York, New York, USA, 2008. ACM Press. 1.1.2, 2.3.1

SHAH et al., 2010. H. Shah, C. Görg and M. Harrold. **Understanding Exception Handling: Viewpoints of Novices and Experts**. *IEEE Transactions on Software Engineering*, 36(2):150–161, Mar. 2010. 1.1.1

SILVA and CASTOR, 2013. T. B. L. Silva and F. Castor. **New exception interfaces for Java-like languages**. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, pages 1661–1666, New York, New York, USA, Mar. 2013. ACM Press. 1.1.1, 1.1.2, 2.3.2, 4.5

SINHA and HARROLD, 2000. S. Sinha and M. J. Harrold. **Analysis and Testing of Programs with Exception Handling Constructs**. *IEEE Transactions on Software Engineering*, 26(9):849, 2000. 1.1.1, 1.1.2, 3.3.1, 4, 4.2.1

SOARES et al. 2002. S. Soares, E. Laureano and P. Borba. **Implementing distribution and persistence aspects with aspectJ**. *ACM SIGPLAN Notices*, 37(11):174, Nov. 2002. 1.1

SWAIN and GUTTMANN, 1983. A. D. Swain and H. E. Guttmann. **Handbook of human reliability analysis with emphasis on nuclear power applications**. Technical report, Sandia National Laboratories, 1983. 3.1.1

TERRA and VALENTE, 2009. R. Terra and M. T. Valente. **A dependency constraint language to manage object-oriented software architectures**. *Software—Practice & Experience*, 39(12):1073–1094, Aug. 2009. 4.5, 4.5

TERRA et al., 2015. R. Terra, M. T. Valente, K. Czarnecki and R. S. Bigonha. **A recommendation system for repairing violations detected by static architecture conformance checking**. *Software: Practice and Experience*, 45(3):315–342, 2015. 1.1.2, 5.6

THUMMALAPENTA and XIE, 2007. S. Thummalapenta and T. Xie. **Parseweb: a programmer assistant for reusing open source code on the web**. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007. 5.1.1

THUMMALAPENTA and XIE, 2009. S. Thummalapenta and T. Xie. **Mining exception-handling rules as sequence association rules**. In *Proceedings of the 31st International Conference on Software Engineering*, pages 496–506. IEEE Computer Society, 2009. 1.1.1, 4, 4.3.2, 4.5

VANDEURSEN et al., 2000. A. van Deursen, P. Klint and J. Visser. **Domain-specific languages: an annotated bibliography**. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. 1.2, 4

VAN OMMERING et al., 2001. R. van Ommering, R. Krikhaar and L. Feijs. **Languages for formalizing, visualizing and verifying software architectures**. *Computer Languages*, 27(1-3):3–18, Apr. 2001. 4.5

ZHAO and ELBAUM, 2003. L. Zhao and S. Elbaum. **Quality assurance under the open source development model**. *Journal of Systems and Software*, 66(1):65–75, Apr. 2003. 3.1.2

FOWLER, 2010. M. Fowler. **Domain-specific languages**. Pearson Education, 2010. 1.2, 4