Enforcing Exception Handling Policies with a Domain-Specific Language

Eiji Adachi Barbosa, *Member, IEEE*, Alessandro Garcia, *Member, IEEE*, Martin P. Robillard, *Member, IEEE*, and Benjamin Jakobus, *Member, IEEE*

Abstract—Current software projects deal with exceptions in implementation and maintenance phases without a clear definition of exception handling policies. We call an exception handling policy the set of design decisions that govern the use of exceptions in a software project. Without an explicit exception handling policy, developers can remain unaware of the originally intended use of exceptions. In this paper, we present Exception Handling Policies Language (EPL), a domain-specific language to specify and verify exception handling policies. The evaluation of EPL was based on a user-centric observational study and case studies. The user-centric study was performed to observe how potential users of the language actually use it. With this study, 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 identified some language characteristics that hindered its use and that motivated new language constructs. In addition, we performed case studies with one open-source project and two industry-strength systems to investigate how specifying and verifying exception handling policies may assist in detecting exception handling problems. The results show that violations of exception handling policies help to indicate potential faults in the exception handling code.

Index Terms—Exception handling, exception handling policy, policy specification, domain-specific language

1 Introduction

Robust programs must provide their functionalities correctly even in the presence of exceptions [24]; for this reason, robust programs usually dedicate part of the source code to exception handling. An exception is any unexpected condition that occurs at runtime and disrupts the normal execution flow of a program [13], [22]. To ease the design and implementation of the exception handling of software systems, most mainstream object-oriented (OO) programming languages provide built-in exception handling mechanisms (EHMs) [13], [22]. These mechanisms provide language constructs to indicate in the source code the places where exceptions are raised and the places where exceptions are handled. Developers can then use these constructs to structure the exception handling of their systems.

To further support the construction of the exception handling part of software systems, some OO programming languages provide EHMs with extra facilities aimed at improving software robustness. These mechanisms are called reliability-driven EHMs [7], [8]. Java is perhaps the best-known programming language that provides a reliability-driven EHM. In Java, exception types can be classified as either *checked* or *unchecked* exceptions. Typically, checked exceptions represent recoverable exceptional conditions, whereas unchecked exceptions represent programming errors, such as accessing an invalid array index [5]. Aiming at improving software robustness, the EHM implemented by Java verifies at compile time if checked exceptions are either handled locally or declared on the signature of methods. Declaring a checked exception on the signature of a method binds the method that calls it to the contract of either handling this exception or also declaring this exception on its signature. If a checked exception is neither handled nor is declared on the signature of the method, then the Java EHM signals an error at compile time. The Java EHM does not perform these reliability checks for unchecked exceptions. For this reason, the declaration of Java unchecked exceptions on the signature of methods is not mandatory. Other programming languages, such as CLU [29] and Eiffel [34], also provide reliability-driven EHMs.

Despite EHMs having been conceived as structuring mechanisms aimed at improving software robustness, failures due to improper use of exception handling constructs are being continually reported in the literature [2], [7], [8], [17], [32], [33], [39]. Not even the use of reliability-driven EHMs seems to improve exception handling, since many exception handling failures are actually reported in systems implemented in programming languages with this type of EHM [2], [7], [17], [32], [33], [39]. One of the problems with exception handling is that developers use exception handling constructs during implementation and maintenance phases without being aware of the exception handling policy of their projects [15], [26]. We call the exception handling policy of a software project the set of design decisions that govern the use of its exceptions. In fact, most software projects currently do not even define an explicit exception handling policy [16]. As a consequence, developers are

- *E.A. Barbosa, A. Garcia, and B. Jakobus are with the OPUS Research Group, Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rua Marquês de São Vicente, 255-Gávea, Rio de Janeiro 22453-900, Brazil. E-mail: {ebarbosa, afgarcia, bjakobus}@inf.puc-rio.br.*
- *M.P. Robillard is with the School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building, Office 114N, Montreal QC H3A 2A7, Canada. E-mail: martin@cs.mcgill.ca.*

Manuscript received 19 Jan. 2015; revised 8 Oct. 2015; accepted 22 Nov. 2015. Date of publication 6 Dec. 2015; date of current version 20 June 2016.
Recommended for acceptance by L. Baresi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2506164

unaware of how they are supposed to use exceptions in their projects, so they end up using them in an *ad-hoc* manner or even neglecting them [40], [41]. Not surprisingly, the exception handling code is more error-prone and contains more faults than the normal code [32], [33], [39].

Currently, there is no proper means to specify, let alone verify, exception handling policies in software projects. Current EHMs assume that only defining the places where exceptions are raised and handled and the list of exceptions a method might throw is enough to define an exception handling policy. However, important aspects of exception handling policies cannot be expressed with current EHMs, not even with the reliability-driven ones. For instance, one cannot define where an exception must be re-mapped to another specific exception type nor the correct place where the exception should be handled. Previous studies performed by Cacho et al. [7], [8] showed that almost 42 percent of exception handling faults observed in industry-strength software systems stem from violations of basic exception handling policies, such as performing re-mappings and leaving the re-mapped exception unhandled. The lack of means to specify and verify explicit exception handling policies in software projects may be one of the main reasons why developers still struggle to implement exception handling.

In this context, we propose the Exception Handling Policies Language (EPL), an external domain-specific language (DSL) [20] for specifying and automatically verifying exception handling policies in Java programs. Exception handling policies are specified in EPL in terms of exception handling rules that must be adhered in the source code. The evaluation of EPL was based on a user-centric observational study and two case studies. We performed a user-centric observational study with ten developers from different organizations to explore how potential users of the language actually use it to produce their exception handling policies. We could observe participants reactions while using EPL and also interview them after they have used the proposed language. With this study, 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. Thus, we identified some language characteristics that hindered its use and that motivated new language constructs. We also performed case studies in the context of one open-source software system—Apache Tomcat—and two different industry-strength software systems—Mobile Media [19] and Health Watcher [28], [44]. For Tomcat, we inspected its architectural specification, extracted from this specification requirements related to exception handling, produced a partial exception handling policy for the system and verified its source code. The violations of the specified policy pinpointed to a severe bug reported in the project's bug tracking system. For the other two systems, we specified their complete exception handling policies and verified the source code compliance. In both systems, we observed considerable amounts of violations of the specified policies. In the context of Mobile Media, we observed violations in 42 percent of the methods handling exceptions. In the context of the Health Watcher, we observed violations in 62 percent of the methods raising an exception. Then, we manually inspected the violations observed in Mobile Media and Health Watcher to explore how these violations can assist developers in finding potential faults of categories of exception handling faults previously proposed by Barbosa et al. [2] and Ebert et al. [17]. We observed that most violations of the intended policy were related to potential faults of the proposed categories of exception handling faults. We also observed violations that may pinpoint to potential faults that were not described in previous work.

The main contribution of this paper is the definition and implementation of EPL, a domain-specific language that allows software designers and developers to make their exception handling policies explicit, helping to raise the awareness of the intended use of exceptions in a software project. We also discuss how violations of specific EPL rules may pinpoint to potential faults of categories of exception handling faults, which can aid developers to proactively detect in their systems potential problems in the exception handling code. The rest of this paper is structured as follows. Section 2 presents background and basic terminology about exception handling mechanisms. Section 3 describes an implementation scenario that serves the purpose of motivating the proposed DSL. Section 4 presents the proposed specification language. Section 6 explains the evaluation procedures and shows the results gathered from the user-centric study and Section 7 presents the case studies performed with industry-strength systems. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2 Exception Handling Mechanisms

Even though exception handling mechanisms are implemented differently across programming languages [21], the vast majority of programming languages with built-in EHM implement the *try-catch* model [25]. The *try-catch* model is depicted in the following generic structure:

```
try{S}
catch(E1 x){H1}
catch (E2 x) {H2}
```

The *try* block delimits a set of statements S. The *try* block guards S from occurrences of exceptions flowing out of the block with a list of exception handlers. An exception handler is defined by a *catch* block in most programming languages.[1] The *catch* block delimits a set of statements H_n, which is responsible for implementing the handling actions that cope with an exception. The *catch* block also declares an argument x of type E_n. The type of this argument serves as a filter to define which types of exception each *catch* block can handle. In most EHMs, if a *catch* block declares as argument type an exception type E, then it can handle exceptions that are instances of E and also exceptions that are instances of subtypes of E. The JavaScript programming language is an exception to this description. Exception handlers in JavaScript do not declare arguments with types and, therefore, do not filter which exception types they can handle. Exception handlers in JavaScript simply capture all exceptions that might be raised within the *try* block.

1. Some programming languages do not use the `catch` keyword for exception handlers. For instance, Python uses the `except` keyword, whereas Ruby uses the `rescue` keyword.

In most programming languages that implement the *try-catch* model, a *finally* block may also be attached to a *try* block. The *finally* block delimits a set of statements that is executed when the control leaves a *try* block. The control can leave a *try* block as a result of normal execution termination (i.e., all statements in the *try* block are executed normally), of the execution of a *break*, *continue*, *go-to* or *return* statement, or of the propagation of an exception flowing out of the *try* 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 given set of statements are accidentally bypassed by a transfer of control. For this reason, *finally* blocks are often used to perform cleanup actions that must be executed even in the presence of exceptions.

2.1 Raising Exceptions and Searching for Handlers

EHMs also provide means to explicitly signal the occurrence of exceptions. Signaling an exception occurrence is structured in most programming languages with the *throw* statement.[2] The *throw* statement causes an exception to be raised. The result of raising an exception is an immediate transfer of control to search for a proper exception handler. The EHM assumes the program control and starts the search from the scope where the exception is raised. If the exception is raised from an unguarded scope (i.e., from a statement outside a *try* block), then the exception propagates up the call stack until a *try* block is found. When a *try* block is found or when the exception is raised from within a *try* block, the EHM searches in the list of *catch* blocks attached to this *try* block for a suitable handler. The runtime type of the raised exception is compared with the exception types in each *catch* block. For the first matching *catch* block, its set of statements is executed. If no matching is found, the exception propagates up the call stack until a matching handler is found. If no handler is found in the whole call stack, the EHM normally terminates the program.

2.2 Propagating Exceptions

If no exception handler is defined in the scope where an exception is raised, the exception can be propagated up the call stack until a proper handler is found. In most mainstream programming languages, the EHM automatically propagates exceptions from where they are raised to where they can be handled [21]. However, there are some programming languages, like CLU [29], that do not support automatic propagation of exceptions [21]. In these languages, exceptions are only propagated between the method who explicitly raised the exception and its immediate caller. To propagate exceptions to higher-level components, the immediate caller has to catch the raised exception and explicitly re-raise it to the next caller in the call stack. In Java, unchecked exceptions are automatically propagated by the underlying EHM. Checked exceptions, on the other hand, are only propagated when they are declared in the method's signature with the *throws* clause. The *throws* clause in Java defines the method's exceptional interface, i.e., the list of exceptions that might occur during the method execution. If a checked exception occurs in the context of a method and it is neither handled nor propagated, then the Java compiler signals a compilation error.

2. Some programming languages, like Ada, Python and Ruby, for instance, use the *raise* keyword to indicate the occurrence of an exception.

2.3 Re-Throwing Exceptions

After being raised, exceptions usually traverse through different intermediate methods on the call stack before they are handled. Intermediate methods are those in an exception propagation path situated between the place where the exception is raised and the place where the exception is handled. In some cases, it might be necessary to implement partial handling actions in intermediate methods. In these cases, intermediate methods need to catch the exception, implement partial handling actions and re-raise the caught exception to its immediate caller. A re-throw occurs when a *throw* statement within a *catch* block raises the exception instance caught by the *catch* block without losing any context information (e.g., the exception stack trace). In C++ and C#, for instance, a re-throw is implemented as follows:

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

In C++ and C#, when a *throw* statement is invoked within a *catch* block and without an argument, the underlying EHM automatically re-raises to the immediate caller the exception instance caught by the *catch* block. In Java, on the other hand, there is no specific command to re-raise an exception. Instead, a re-throw is performed as follows:

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

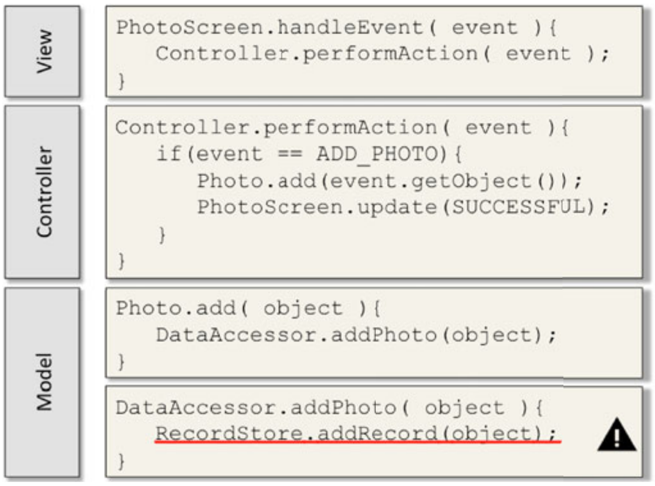
In Java, a re-throw is implemented with a *throw* statement within the *catch* block that explicitly receives as its argument the same exception instance captured by the *catch* block. Although syntactically different, both re-throws are semantically equivalent, since they both re-raise the exception caught by the *catch* block without losing any information.

2.4 Re-Mapping Exceptions

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

```
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 instances of E_1 and subtypes of E_1 and raises an exception instance of type E_2. In this case, it is said that the exception is



```
PhotoScreen.handleEvent( event ){
   Controller.performAction( event );
}
```

```
Controller.performAction( event ){
   if(event == ADD_PHOTO){
      Photo.add(event.getObject());
      PhotoScreen.update(SUCCESSFUL);
   }
}
```

```
Photo.add( object ){
   DataAccessor.addPhoto(object);
}
```

```
DataAccessor.addPhoto( object ){
   RecordStore.addRecord(object);
}
```

Fig. 1. Inserting a photo in mobile media.

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 and C# built-in exception types have constructors that take an exception instance 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.

3 Motivating Example

We present a typical scenario involving exceptions in this section. The scenario serves the purpose of illustrating the need for exception handling policies. Consider the following scenario depicted in Fig. 1, which was extracted from the Mobile Media application [19]. Mobile Media is a mobile application responsible for managing albums of images, videos and audio files in mobile devices running Java Micro Edition. The Mobile Media architecture adheres to the *Model*-View-Controller architectural pattern.

3.1 The Intended Exception Handling Policy

In the example depicted in Fig. 1, when the `RecordStore.addRecord` method is executed, it might throw exceptions of the `RecordStoreException` type. An instance of the `RecordStoreException` type represents an exceptional condition that occurs when accessing the device storage system. Therefore, if this exception occurs, it is not possible to add the selected photo to a given album and there is nothing else that could be done to persist the photo in the device. Hence, the application should warn its user about the problem that prevents storing a given image on the device. To achieve this requirement, it is necessary to decide how the exception should be handled. To warn the user about the problem, it is necessary to propagate the exception from the scope where it occurred (within the *Model* module) up to the *Controller* module scope, so that it handles the exception and updates the *View* with an error message. However, directly propagating instances of `RecordStoreException` from the scope of the *Model* module to the scope of *Controller* module would break the information hiding principle, since the exception contains information pertaining to the implementation of the *Model* module that it is more than what is permitted by its encapsulation. To avoid this, when `RecordStoreException` crosses the boundaries of the *Model* and *Controller* modules, it should be re-mapped to a more abstract exception type called `PersistenceException`. Thus, this more abstract exception type re-mapped by the *Model* module provides sufficient information about the cause of the problem, but does not expose implementation details. In addition, the *Controller* module is able to identify the cause of the problem and has proper information to handle the exception. In this manner, it is possible to address the requirement (showing an error message) without breaking any design principle (e.g., information hiding). This set of decisions comprises the intended use of exceptions in this scenario.

3.2 Implementing Exception Handling in Practice

The complete intended exception handling policy is difficult to discover and understand from the source code. The exception handling code is usually scattered throughout the whole system's code and tangled with different functionalities of the system, hindering its inspection and comprehension. If we take into account all exception types that occur in the source code of Mobile Media, there are 372 references to exceptions in the source code. These references occur in 30 different classes, accounting for almost 3.0 KLOC. Moreover, important parts that define an exception handling policy cannot be defined in the source code with the exception handling constructs of current programming languages. For instance, the exception handling policy previously described defines that instances of the `RecordStoreException` type must be re-mapped to the more abstract type `PersistenceException` by the *Model* module and that the *Controller* module must handle instances of the `PersistenceException` type. These exception handling responsibilities are fundamental parts of the intended exception handling policy, but cannot be expressed in the source code with the EHMs provided by current OO programming languages.

By only analyzing the source code, it is not possible to discover what parts of the intended exception handling policy are not expressed in the source code. In addition, it is also not possible to know if the implemented exception handling code adheres to the originally intended exception handling policy or if it actually violates it. Unaware of how exceptions are supposed to be used, developers tend to ignore exceptions [40], [41]. They either opt to leave exceptions unhandled or they catch them in the scope where they occurred with an empty *catch* block. In the context of the code snippet depicted in Fig. 1, ignoring the occurrence of `RecordStoreException` would introduce a fault in the application. In the first case, there would be no handler for the exception. So if the exception occurs, it is propagated up to the program execution entry point, causing the program execution to be terminated by the underlying EHM. In the second case, the photo would not be added to the device due to the exception, no error message would be shown to the user, but the application would continue to execute normally, updating the screen with a successful message (invoking the `PhotoScreen.update` method as can be observed in the code snippet depicted in Fig. 1). Similar scenarios are commonly observed in industry-strength systems [2], [7], [8], [17], [32], [33], [39].

4 Making Exception Handling Policies Explicit

The Exception Handling Policies Language is an external domain-specific language [20] aimed at supporting the explicit definition of exception handling policies. In object-oriented programming languages with built-in EHMs, exceptions are used at the method level, i.e., methods are the source code elements that raise, handle, propagate, re-throw or re-map exceptions. However, defining a system's exception handling policy by enumerating the exception handling responsibilities of its methods would not scale well, since systems have a large number of methods. To specify exception handling policies at a higher level of abstraction, EPL provides the concept of *Compartment*. In EPL, a compartment is a named and referable set of methods to which responsibilities can be specified. This concept is further discussed in Section 4.1.

In EPL, the *Rule* concept is used to express responsibilities. These responsibilities are expressed in terms of permissions, prohibitions and obligations about how compartments and specific exception types can establish dependency relations. The language supports dependency relations originated from exceptions being raised, handled, propagated, re-thrown and re-mapped. These relations cover all structural dependencies between source code elements and exception types common in programming languages with built-in EHMs (Section 2). The rule concept is further discussed in Section 4.2. Finally, in Section 4.3 we present the *Alias* concept, which allows the definition of an alias for a list of exceptions in the specification of exception handling policies.

4.1 Compartments

In EPL, a compartment is a named and referable set of code elements that can relate to exceptions in the source code. In an object-oriented programming language, for instance, a compartment would be a set of methods. In EPL, 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 "*" can be used to define a name pattern. The following example depicts how a compartment can be defined at a fine-grained level:

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

In the previous example, the *DATA-ACCESS* compartment comprises all code elements of 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 code elements of all modules whose fully qualified names have the prefix `pucrio.controller`. In Java, for instance, it would comprise all methods of all classes whose fully qualified names have 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 code elements of the 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 IDataAccessor;
```

In the previous example, the *DATA-ACCESS* compartment comprises all code elements of the modules that are subtypes of the `IDataAccessor` type and whose fully qualified names have the prefix `create`, `read`, `update` or `delete`. In Java, for instance, this compartment would comprise the methods whose names start with either `create`, `read`, `update` or `delete` of the classes that are subtypes of the `IDataAccessor` type. 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 7).

4.2 Rules

The main purpose of an exception handling policy is to explicitly define which responsibilities source code elements have to comply with specific exception types. These responsibilities are expressed in EPL in terms of permissions, prohibitions and obligations. More specifically, the rule concept expresses permissions, prohibitions and obligations about how compartments can establish dependency relations with specific exception types. We consider that a given compartment C establishes a dependency relation with a specific exception type E if there exists one of C's element that establishes a dependency relation with an exception instance of the type E. Moreover, we consider that dependency relations 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" dependency relations between exceptions and code elements, since they are

TABLE 1
Dependency Relation Types

Dependency type	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 type $E2$ in its scope
m re-throws E	Method m re-throws an exception of type E from its scope

typical relations with which developers structure their exception handling code (see Section 2). Table 1 summarizes these dependency relation types.

EPL provides two rule types to express permissions: the *Only-May* rule type and the *May-Only* rule type. Rules of the *Only-May* type express permissions about which compartments can establish dependency relations 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 a dependency type (`<dep_type>`), which may be one of the dependency types shown in Table 1 and a list of exception type identifiers (`<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 dependency types.

The other 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 dependency relations. This rule type is expressed with the following syntactic structure:

`<comp_id>` **`mayonly`** `<dep_type>` `<exception_list>`

The *May-Only* rule type is syntactically similar to the previous rule type. It is also defined in terms of a dependency type and a list of exception type identifiers. 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`** `re-map` **`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 remapping 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 dependency relations.

Notice in the previous example that the re-map dependency type 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 dependency relation 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` $\texttt{E}_1$ `to` $\texttt{E}_2$. This syntactic structure for the re-map dependency relation is the same for the other rule types.

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 a dependency relation and a list of exception type identifiers 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 dependency relations.

EPL also provides a rule type to express prohibitions. The *Cannot* rule type is used to express rules that prohibit compartments to establish dependency relations 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 6). 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 dependency relations.

4.3 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 6). 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.

5 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 if 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. In Section 5.1 we detail the *Facts Extractor* module and in Section 5.2 we detail how the *Rule Checker* works.

5.1 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 target system is parsed and its Abstract Syntax Tree (AST) is analyzed in order to extract dependency facts related to the exception handling dependency relations supported by EPL. To extract the dependency facts related to the dependency relations supported by EPL, which are *Handle*, *Propagate*, *Raise*, *Re-map* and *Re-throw* 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* dependency relations. The *Facts Extractor* module registers only one dependency fact for each *catch* block. The following pseudo code 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 pseudo code, 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 instance caught 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 we consider 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 dependency relation; they may be related to the *Raise*, *Re-map* and *Re-throw* dependency relations. The following pseudo code 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 of a the *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 ){
     log(e);
  } 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* dependency relation. Thus, the module registers that the `foo()` method establishes *Propagate* dependency relations with the `IOException`, `MyException2` and `MyException3` types. It should be noted that *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 `FileNotFound-Exception`. To specify which exceptions *throw* statements should raise, one should use *Raise* rules. For this reason, the dependency facts related to the *Propagate* dependency relation 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 structure 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 [1], [7]. 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 and, as such, needs to be specified and verified throughout the software development process. For this reason, we believe that it is more important to define which exception types are allowed to be declared on the exceptional interfaces of Java methods, rather than to define which specific exception types are allowed to flow through the boundaries of methods.

In addition, EPL does not support the specification of rules describing 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, we focus on supporting the specification and verification of rules related to the *Propagate* dependency type only in terms of the *throws* clause of Java methods. This way, 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. There are also innumerable 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 significant burden is one of the reasons of why unchecked exceptions are not required to be declared in exceptional interfaces in Java [5].

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 relation 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 relation 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 instance of the `MyException3` type to another exception instance 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. Also, we consider that re-mappings may occur between exception instances of different types and also between exception instances of the same type, as shown in the previous example.

To extract the dependency facts related to the *Raise* and *Re-map* dependency relations, 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. For 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 relation with the `FileNotFoundException` type. However, when the expression of *throw* statements refers to variables, 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 [43]. 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 `MyExcep-tion`, 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 too. 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 *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* dependency relations with both `Type1Exception` and `Type2Exception`. 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. 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 always 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 [43], so in most cases the types of raised exceptions can be precisely determined without loss of precision.

5.2 Checking the Rules

The *Rule Checker* module checks for each specified rule in the exception handling policy if there exists 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 a dependency relation to a list of exception types. Consider a rule R that specifies how a compartment C is allowed to establish a dependency relation D to a list E of exception types. A violation to a *Cannot* rule is defined as follows:

$$\exists m\texttt{:Method} \in C \wedge \exists e\texttt{:Exception} \in E \mid D(m, e)$$

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

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

$$\exists m\texttt{:Method} \in C \wedge \exists e\texttt{:Exception} \notin E \mid D(m, e)$$

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

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

$$\exists m\texttt{:Method} \notin C \wedge \exists e\texttt{:Exception} \in E \mid D(m, e)$$

A violation to a rule R of the *Only-May* type occurs when there exists a method m not in compartment C establishing a dependency relation 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 a dependency relation D with the exception types in E, but a method not in C is establishing a dependency relation D with an exception type in E.

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

$$\exists e\texttt{:Exception} \in E \wedge \nexists m\texttt{:Method} \in C \mid D(m, e)$$

A violation to 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 a dependency relation 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 a dependency relation D with e.

Verification warnings. EPL makes it possible to define sets of inconsistent rules; that is, 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*.

5.2.1 Conflict between Cannot and Must

Given the same compartment and the same dependency type, 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;
```

5.2.2 Conflict between Only-May and Only-May

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

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

5.2.3 Conflict between Must and May-Only

Given a common compartment, a *Must* rule and a *May-Only* rule will conflict if an exception type 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;
```

5.2.4 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 first 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.

6 User-Centric Evaluation

After designing the EPL language, we designed a study to explore how developers use EPL to produce their exception handling policies. In addition, we were also interested in exploring if the language actually provides suitable constructs for specifying exception handling policies. To explore these design dimensions, we needed to observe the use of EPL and also to gather the reactions of developers that used EPL. In this context, we performed an exploratory user-centric observational study. 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 use the observational study to give participants context, i.e., we could set up scenarios in which participants could perform tasks using EPL while we directly observed them. Second, we could use the interview to gather participants' reactions about their experience in using the language. In the next sections we detail the settings of our observational study (Section 6.1) and we present its results (Section 6.2).

6.1 Study Design

The main goal of this study was to observe how participants use EPL to produce their exception handling policies. To do so, we set up tasks to expose participants to the use of EPL. The sessions of the observational study were performed individually. Each session comprised two tasks of 30 minutes each. The goal of the tasks was to mimic different scenarios of use of the language. Thus, we could also observe how participants use 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 implemented exception handling policy from the source code. The goal of participants was to inspect the source code of the target system and extract the implemented exception handling policy. Participants received only the source code of the target system in this task. 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. 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 [15], [26].

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 [6]. In this task, participants received the system documentation. The system documentation describes the intended architecture of the target system, showing its main components in a component diagram, as well as the intended relations between these components. The documentation also describes the intended exception hierarchy tree, showing the hierarchy structure in a class diagram. Finally, the documentation describes 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 15-20 minutes) about: basic exception handling concepts, the concepts provided by the proposed specification language, the study settings and the architecture of the target system. The presentation covered all main topics of EPL: compartments, types of rules and dependency types. Participant 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. In this manner, participants were exposed to all main features of EPL. 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 if participants would be aware of possible conflicting rules. During the study, the researcher only observed the participants and did not participate in the task. Questions about concepts and the use of the language or the target system were answered with *"you can consult the provided documentation"*. After the lecture and before actually performing the first task, participants were allowed to read the document describing the language without a limit of time. The artifacts used to perform the observational study and the interviews are available on-line.[3]

6.1.1 Data Collection and Data Processing

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 first author 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 questions to be asked during the post-task interview.

We analyzed the collected data with a mix of quantitative and qualitative methods [12]. The specifications produced by participants were saved at the end of each task and later analyzed by the researchers. We quantitatively analyzed each specification produced by computing its size in terms of the number of compartments and rules specified. We also qualitatively compared the specifications of each task in terms of how compartments are defined and in terms of what rule types and dependency types are used.

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 researchers in order to report the results. To analyze the collected data, we adopted an iterative coding process. First, we extracted the main fragments of participants answers and assigned a topic to each of these fragments. In this context, 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 if 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.

6.1.2 Target System

To produce the artifacts required to perform the study, we needed a software system for which we had access to its source code and its intended exception handling policy. We used the Mobile Media system as the target system for three main reasons. First, Mobile Media is a well documented product line that has been used in previous empirical studies in exception handling [9], [11], [38]. Second, we could contact the original designers of Mobile Media to elucidate its intended exception handling policy. Third, as Mobile Media is a product line, the general exception handling policy would need to be specified in a way that it would be reused and applicable to all its more than 100 products. 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 consulting the source code and existing artifacts of Mobile Media and with the help of its original designers.

3. Available at: http://www.les.inf.puc-rio.br/opus/EPL

TABLE 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

6.1.3 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. We selected the set of participants of our study in an iterative manner: we selected one new participant, performed the observational study followed by the interview and analyzed the collected data. We repeated this iteration until we reached saturation, i.e., until we analyzed the collected data and observed that the emerging topics were already previously mapped. We selected ten participants, although we reached saturation with eight. Participants in our study had their university education in different institutions from 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 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 exceptions 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 different exception handling mechanisms. For instance, in Java, the compiler automatically verifies if there exist proper handlers for checked exceptions. In C#, JavaScript and PHP, on the other hand, there exist exceptions, but no automatic verification for proper handlers. Thus, participants were not biased towards a specific exception handling mechanism.

6.1.4 Interview Guide

The goal of the interview was to gather participants' reactions about their experience in using the specification language. To help us in understanding participants usage of the EPL language we relied on a technology adoption model that try to explain the determinant factors of technology usage behavior. Thus, we built our interview guide based on the Technology Acceptance *Model* (TAM) [14]. 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 dimension 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. Thus, we would have flexibility to explore unforeseen information that could emerge during the interviews.

6.2 Results

This section presents the results of our observational study. In Section 6.2.1 we present the analysis of the specifications produced in each task of the observational study, whereas in Section 6.2.2 we present our observations of how developers used the language in each task. Finally, in Section 6.2.3 we present the analysis of the interviews.

6.2.1 Artifacts Analysis

Comparing the specifications produced during the first task, we could observe that participants produced specifications with similar compartments, but with very different rules. As can be observed in Table 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 of the target system comprised five high-level packages: *Controller*, *Screen*, *AlbumData*, *ImageAccessor* and *ImageUtil*. We observed in the specifications 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 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. Thus, participants did not use it in this study.

TABLE 3
Specifications Produced in the First Task

	# Compartments	# Rules
Min	2	2
Max	6	30
Average	4	7.2
Median	4	4.5

For the specified rules, we can observe in Table 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 two to nine rules. While participant P2 was performing the first task, we observed that for the rules that could be specified in terms of a list of exceptions, he specified one rule for each exception type 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 used rules using exception lists, his specification would have nine rules, instead of 30. It is worth noting 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 dependency type used, some specifications produced in the first task were defined in terms of only one dependency type: participants P3 and P5 specified only rules of the *Handle* dependency type, whereas participant P7 specified only rules of the *Raise* dependency type. The specification produced by the participant P2 was the only one to cover all the exception handling dependency types provided by EPL. The specifications produced by the other participants comprised no more than two different dependency types, 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

TABLE 4
Specifications Produced in the Second Task

	# Compartments	# Rules
Min	0	6
Max	4	21
Average	3.4	8.4
Median	4	7

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 the specifications produced were consistent with the exception handling code implemented in the target system. That is, we checked if the specified rules referred to dependency relations and exception types that exist 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 order of how participants should produce their specifications. This decision implies 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 adheres to an exception handling policy and have produced an idealized exception handling policy that should have been followed in the target system. 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. We consider it only as an indicator of the different approaches adopted by participants in this task.

For the specifications produced in the second task, we can observe in Table 4 that there was one specification with 0 compartments definitions, 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 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 with 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 six rules to a maximum of nine rules.

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

> *The following exceptions are raised by third party APIs and are handled in the context of the Image Accessor 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 mistaken the specification of a rule related to the same requirement; they forgot to specify some exception types in a given rule. This requirement was specified in two different parts of the system documentation. Thus, participants may have missed 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 exception types that were not defined in the system documentation and these types did not seem to represent cases of other existing types 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 four compartments comprising the same components. In fact, we observed that participants P4 and P8 initially specified four 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.

Analyzing the artifacts produced in the second task, 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 Re-map* 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 from the lapse of participant P9 in the second task, we did not observe any other serious errors in the specifications produced in both tasks. The most recurring error observed was the lack of the *"from"* keyword in the rules of the *Re-map* dependency type. A total of six out of the 10 participants forgot this keyword in their specifications. In addition, the other errors we found were minor syntax errors, such as a missing semicolon or a misspelled keyword.

6.2.2 Observation Analysis

By observing how participants used the language during the observational study, we could observe distinct approaches in each 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. In particular, we could observe that some participants adopted systematic approaches to inspect the source code, whereas others seemed to inspect the source code at random. Moreover, we could observe 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 dependency type, the *Handle* and *Raise* dependency types, respectively. This is actually aligned with their systematic approach of searching for keywords related to specific exception handling dependency relations. Participant P2, on the other hand, was the only participant to produce a specification covering all exception handling dependency types 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 relation. 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.

During the second task, when participants had to extract the exception handling policy from 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 a given exception handling rule. 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 from 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.

6.2.3 Interview Analysis

We extracted 133 fragments from the interviews transcriptions and a total of six topics emerged from these fragments. 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). The results of this study take the form of quotes that illustrate the participants' reactions about different dimensions of EPL captured by the six topics.

a) Perceived usefulness. The *Perceived Usefulness* topic groups together the fragments in which participants mention if and in which ways the EPL would be useful to them. Overall, all participants considered the proposed DSL 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.

Other participants perceived the language as a useful means to support software quality assurance practices. Participants P2 and P4 mentioned that using the proposed DSL would be useful to support software inspections. About this P2 mentioned:

P2: In an inspection meeting there would be probably less anomalies (related to exceptions) to fix. Then, 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!"

Participants P1 and P10 mentioned that using the language would prevent defects and bad programming practices related to exception handling. Participant P1 mentioned:

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.

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.

Other participants mentioned more 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.

b) Expressiveness. The *Expressiveness* topic groups together the fragments in which participants commented on if 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, some participants also mentioned issues with the language that hinders its expressive power. In particular, six 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 type 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, as described in Section 4, is an improvement that was incorporated *a posteriori* into EPL.

c) 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 in Sections 6.2.1 and 6.2.2. During the interviews, all participants answered that EPL is easy to use. However, some participants pondered that defining the exception handling policy was not easy, especially 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, especially 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.*

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 in Section 4. Finally, it is worth noting that during the interviews no participant commented on the possibility of specifying inconsistent rules.

d) 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" simply do not exist. 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. He believed that the verification tool of the DSL would yield many warnings about violations. 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.*

e) 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 pinpointed 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.

f) Comprehensibility. The topic *Comprehensibility* groups together the fragments in which participants commented on if 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 names since they have similar semantics, i.e., they both express permissions. Although we understand that this kind of confusion may occur to other users, we do not plan to change the keywords used for each rule. Moreover, when questioned about the semantics of the rule types, P1 confirmed that he understood it; he only hesitated to associate the semantics to the rule type name.

6.3 Limitations of the Study

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. Thus, 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.

Another limitation of the study relates to the target system used in the observational study, especially due 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 DSL. Another limitation of our study is the possible cultural-related bias of participants. To mitigate this possible threat, we selected participants from different organizations, who had their university education in different institutions and with different backgrounds. Thus, the specific organization culture or specific education training was not a significant threat to our study validity.

Finally, our interview guide has probably not covered every important question that could have been asked to participants. We built our interview guide based on the Technology Acceptance *Model* [14], which is an empirically tested model to study factors that influence technology adoption. In this manner, we tried to cover in our interview guide important factors that were tested in previous empirical studies. Moreover, we organized semi-structured interviews in such a way 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 the participants were sharing interesting information about topics not covered in our interview guide.

7 Case Studies

In Section 4, we presented EPL, a DSL for specifying exception handling policies. We argued that the lack of exception handling policies may be one of the reasons why exception handling code is more error prone and contains more faults than the overall code. In this study, we aimed at investigating if and to what extent verifiable exception handling policies can assist developers in finding exception handling faults. Our aim is stated in the following research question:

RQ: How do verifiable exception handling policies assist in the detection of exception handling faults?

To answer our research question, we assessed EPL in two different settings. In the first setting, we specified the exception handling policy of an open-source software system and verified its source code. Then, we inspected the violations of the specified policy to check if they were related to reported failures of the system. We present the details of this case study in Section 7.1. In the second setting, we employed EPL to specify the complete exception handling policy of two industry-strength systems and verified their source code. Then, we analyzed the violations of the specified policies to assess how these violations relate to potential faults of categories of exception handling faults previously proposed by Barbosa et al. [2] and Ebert et al. [17]. We detail this second case study in Section 7.2.

7.1 Partial Specification Analysis

In this assessment, we employed EPL to specify parts of the Apache Tomcat web server. 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 requirements from the JEE architecture specification. Second, Tomcat is a widely adopted open source project with source code and bug reports repositories publicly available. Thus, we could use EPL to check if Tomcat implementation complies with the exception handling requirements specified in the JEE architecture specification and we could also check if the violations observed were related to reported failures of the system.

The JEE architecture specification is described in the Java Specification Request—JSR—number 342.[4] 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

4. Available at: https://jcp.org/en/jsr/detail?id=342

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. Thus, JSRs can be trusted as reliable specifications of Java technologies.

To extract requirements related to exception handling from JSR-342 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 components implemented by Tomcat; the other three were specified for components of the JEE architecture that are not implemented by Tomcat.

The first exception handling requirement specified for components 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, page 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 sub-
        type 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 type.

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, page 174)*

The second exception handling requirement also expresses an obligation, but in a different context and for a different exception type. 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 compart-
        ment SERVLET-SESSION where X is subtype
        of javax.servlet.HttpSession;
SERVLET-SESSION      must      raise      java.lang.
        IllegalArgumentException;
```

The elements of interest related to the Requirement 2 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 if 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. We inspected Tomcat issue tracking system and discovered that this violation was already reported as a critical bug.[5] 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:

> *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

5. Available at: https://bz.apache.org/bugzilla/show_bug.cgi?id=51744

TABLE 5
Exception Handling Policy Violations

Target System	Site Type	Total	With Violations
Mobile Media	Handling Site	63	27
	Propagation Site	88	0
	Raising Site	9	2
	Re-mapping Site	19	4
	Re-throwing Site	0	0
Health Watcher	Handling Site	197	26
	Propagation Site	1,064	3
	Raising Site	99	65
	Re-mapping Site	185	20
	Re-throwing Site	78	1

among six 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 requirement defined for the JEE architecture and identified violations in the source code. These violations pinpointed directly to the causes of the reported bug. Thus, Tomcat's verifiable exception handling policy was able to detect a severe fault in the exception handling code.

7.2 Complete Specification Analysis

In this assessment, we specified the complete exception handling policy of the systems Mobile Media v.9 and Health Watcher v.10. We used these systems as the target systems for two main reasons. First, they are well documented systems that have been used in previous empirical studies in exception handling [9], [11], [38]. Second, we could contact their original designers to validate the intended exception handling policy.

The Mobile Media architecture adheres to the *Model*-View-Controller (MVC) architecture pattern. Mobile Media's exception handling policy aligns with its architecture, with compartments defined for each module of the MCV 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 and propagating these exceptions to *Controller*. The *Domain* and *View* modules are not allowed to handle or raise any exceptions. The complete exception handling policy specification produced for Mobile Media v.9 comprises four compartments and 18 rules definitions.

The Health Watcher is a web-based system for registering complaints about health units. It has a layered architecture, with the layers: *GUI*, *Business*, *Data* and *Persistence*. It also has an independent module responsible for dealing with *Distribution* and a *Façade* module that manages the communication between *Business* and *GUI* and between *Distribution* and *GUI*. Health Watcher's exception handling policy also aligns with its architecture, with layers and compartments having the same boundaries. Each compartment has the following responsibilities. *Persistence* and *Distribution* are responsible for re-mapping API-specific exceptions to application-specific exceptions. *Business* is responsible for re-mapping some persistence exceptions to business exceptions, whereas the other persistence exceptions are propagated by *Business* and handled by *Façade*. *GUI* is responsible for handling business and distribution exceptions. *Data* is not allowed to handle or raise any exception; it only propagates exceptions from *Persistence* to *Business*. The complete exception handling policy specification produced for Health Watcher v.10 comprises six compartments and 20 rules definition.

After producing the intended exception handling policy for the target systems, we used the *EPL Verifier* to check whether the source code of the target systems adhered or not to their policies. From a total of 18 rules specified for the Mobile Media, five different rules were violated in the source code. For the Health Watcher, from a total of 20 rules specified, eight different rules were violated in the source code. Moreover, each rule can be violated more than once. Thus, the total number of violations observed in the source code is presented in Table 5.

Table 5 presents the total number of handling, propagation, raising, re-mapping and re-throwing sites observed in each system, as well as the corresponding number of violations observed in each site type. A handling site is a method in the source code where an exception is handled. A violation in a handling site means that a specific method establishes a *Handle* dependency relation that it is not allowed to, or that a specific method does not establish a *Handle* dependency relation that it is supposed to. These definitions hold similarly for the other site types.

In the context of the Mobile Media target system, a total of 33 violations were observed, where 27 violations were observed in handling sites, two violations were observed in raising sites and four in re-mapping sites. In the context of the Health Watcher target system, a total of 115 violations were observed, where 26 violations were observed in handling sites, three violations in propagation sites, 65 violations in raising sites, 20 violations in re-mapping sites and one violation in re-throwing site.

Unlike the Tomcat target system, the Mobile Media and Health Watcher target system did not have publicly available bug reports, so that we could assess if the violations observed were related to reported failures in these systems. For this reason, we manually inspected each violation observed in the Mobile Media and Health Watcher target systems to assess if and how they were related to potential faults of categories of exception handling faults previously defined by Barbosa et al. [2] and by Ebert et al. [17]. We refer to these violations as being related to "potential" faults because they are very similar to the descriptions of the categories of exception handling faults proposed by Barbosa et al. and by Ebert et al., but we could not confirm if they were actual faults, since there were no bug reports available. Next, we discuss the manual inspection of the detected violations in both target systems.

7.2.1 Violations in Handling Sites

From the 27 handling site violations observed in Mobile Media, there were five violations of *Only-May Handle* rules, three violations of *May-Only Handle* rules and 19 violations of *Cannot Handle* rules. From the 26 handling site violations observed in Health Watcher, there were four violations of *Must Handle* rules, six violations of *Only-May Handle* rules and 16 violations of *May-Only Handle* rules.

Violations to *Must Handle* rules mean that a given compartment is obligated to handle exceptions of a given type, but it does not handle exceptions of this type. A similar definition is given by Ebert et al. to the category named *"Lack of a Handler That Should Exist"*, which, as the name suggests, occurs when a given module is supposed to handle an exception, but it does not handle it. Thus, violations to *Must Handle* may pinpoint potential faults of this category.

Violations to *Only-May Handle* rules mean that a given compartment is handling exceptions that should be handled by another compartment. A similar definition is also given by Ebert et al. to the fault category named *"Exception Caught at the Wrong Level"*. Faults of this category occur when the exception is handled in a place that is not the one intended by the developers of the system. Given the similarity of these two definitions, violations to *Only-May Handle* could be used to pinpoint potential faults of the *"Exception Caught at the Wrong Level"* category. In addition, specific scenarios of violations of *May-Only Handle* and *Cannot Handle* rules may also pinpoint to potential faults of this category. When a given compartment handles an exception of a type that it is not allowed to handle and another compartment is allowed to handle this same exception type, then this may suggest that this exception is being handled in the wrong compartment. Therefore, this type of scenarios of violations could also be used to pinpoint potential faults of the *"Exception Caught at the Wrong Level"* category.

Finally, there were some handling site violations that were capturing exceptions with a *catch* block whose argument is the `Exception` type. We observed 15 of these violations in Mobile Media and 20 in Health Watcher. Barbosa et al. defined the category of faults named *"Overly Generic catch block"* and Ebert et al. defined an equivalent category named *"General catch block"*. These categories comprise faults that occur when a *catch* block has as argument an exception with an overly-generic exception type, inadvertently catches exceptions by subsumption and leads the system to an unexpected and erroneous state. In EPL, violations to rules *Cannot handle X*, where X is a list of overly-generic exception types (e.g., `Exception, Throwable`), could be used to pinpoint potential faults of these categories. The following code snippet exemplifies a potential fault introduced by a generic *catch* block in Mobile Media:

```
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 *View* compartment of Mobile Media. The stop method is an implementation of the interface `javax.microedition.media.Player` and may throw instances of `javax.microedition.media.MediaException` and also instances of `java.lang.IllegalStateException`. The documentation of the stop method specifies that instances of `MediaException` may be thrown if the `Player` cannot be stopped. The documentation also specifies that instances of `IllegalStateException` may be thrown if the Player is closed. Thus, if the invocation to the stop method in the previous code snippet throws a `MediaException`, the generic *catch* block captures it and only prints its stack-trace. As a consequence, the player is not actually stopped and the user of the application is not informed about the problem. Then, the user of the application will observe a failure in the system.

In Mobile Media, there was also 1 handling site violation implementing a *catch* block whose argument is the `RuntimeException` type, which can also be considered an overly-generic exception type. In fact, handling exceptions that are instances of the `RuntimeException` type is a discouraged practice in Java [5], since these type of exceptions typically represent programming errors that client code cannot do anything to recover from.

7.2.2 Violations in Raising Sites

In Mobile Media, the two raising site violations occurred because a rule of the *Cannot Raise* type was violated. In Health Watcher, from the 65 raising site violations, there were 10 violations of *May-Only Raise* rules, 12 violations of *Must Raise* rules and 43 violations of *Only-May Raise* rules.

Violations to *Cannot Raise* and *May-Only Raise* rules mean that a given compartment raises an exception of a type that it is not allowed to raise. Violations to these rules may pinpoint to potential faults categorized as *"Exception That Should Not Have Been Thrown"*. Ebert et al. define this fault category as the category of faults that usually occur when part of a program should not try to detect the error, possibly because it will be detected somewhere else. Similarly, violations to *Only-May Raise* rules mean that a given compartment raises an exception that is supposed to be raised somewhere else by another compartment. For this reason, violations to *Only-May Raise* may also pinpoint to potential faults of this category.

Violations to *Must Raise* rules mean that a given compartment does not raise exceptions of a type that it is expected to raise. We observed that violations to *Must Raise* rules may pinpoint to two different categories of faults. If a given compartment is obligated to raise a given exception of type X, but it does not raise any exception, then this may pinpoint to potential faults of the category named *"Exception Not Thrown"*. This category, as the name suggests, comprise faults caused by modules not raising exceptions that they are expected to raise. If, on the other hand, a given compartment is obligated to raise a given exception of type X, but it only raises exceptions of other types, then this may pinpoint to potential faults of the category named *"Throwing Wrong Type"*. This category, as the name suggests, occurs when a given module raises an exception, but it uses the wrong exception type. Faults of the category named *"Throwing Wrong Type"* could also be pinpointed by violations of

Cannot Raise and *May-Only Raise* rules, since these rule types define the exception types that a given compartment is allowed to raise.

One of the categories of exception handling faults related to raising site is the *"Uninformative Generic Type Thrown"* category, which is defined by Barbosa et al. as the category of faults caused when an exception with an overly generic type is thrown and, therefore, module clients cannot implement proper handling actions. In the context of Mobile Media, we observed two raising site violations caused because the `Exception` type was raised, whereas in Health Watcher, we observed three raising site violations caused because the *RuntimeException* type was raised. Thus, violations to *Cannot Raise X* rules, where *X* is a list of generic exception types, may explicitly pinpoint to potential faults of this category.

7.2.3 Violations in Re-Mapping Sites

From the categorizations of exception handling faults proposed by Barbosa et al. and Ebert et al., there is only one fault category related to re-mappings. Barbosa et al. proposed the *"Destructive Remapping"* category, which occurs when information is lost during the remapping process. For the re-mapping observed in Mobile Media and Health Watcher, we could not confirm if the re-mapping performed were actually losing information or not. We could only observe that some re-mappings did not wrap the caught exception in the re-mapped exception, although re-mappings that do not wrap the caught exception in the re-mapped exception are not necessarily *"Destructive Remapping"* faults.

In the context of Mobile Media, we observed one violation to a *Must Re-map* rule and three violations to *Cannot Re-map* rules. In the context of Health Watcher, we observed three violations to *Must Re-map* rules, six violations to *Only-May Re-map* rules and 11 violations to *May-Only Re-map*. We also observed in Health Watcher 14 re-mapping violations implementing *catch* blocks declaring the *Exception* type, as well as four re-mapping violations re-mapping the caught exception to the *RuntimeException* type. The following code snippet extracted from Health Watcher exemplifies these violations:

```
public  long  searchTimestamp(String  table,
        String id){
  try {
    PersistenceMechanism pm =
    PersistenceMechanism.getInstance();
  (...)
    return answer;
  } catch (Exception ex){
    throw new RuntimeException();
  }
}
```

In the previous code snippet extracted from the *Persistence* compartment of Health Watcher, the `getInstance` method may raise an exception instance of the `RepositoryException` type. The exception instance of the `RepositoryException` type is captured by subsumption by the generic *catch* block and re-mapped to the `RuntimeException` type. The first violation observed in the previous code snippet is that exceptions of the `RepositoryException` type are not supposed to be re-mapped at the *Persistence* compartment; they are supposed to be propagated from the *Persistence* compartment to the *View* compartment, where they are supposed to be handled. The other violation observed in the previous code snippet is that the *Persistence* compartment is not allowed to remap from the `Exception` type to the `Runtime` type. Although these violations are not defined by Barbosa et al. and Ebert et al. as exception handling fault categories, they resemble some other fault categories, such as: *"Exception Caught at the Wrong Level"*, *"Overly Generic catch block"* and *"Uninformative Generic Type Thrown"*. In addition, previous studies from Cacho et al. [7], [8] showed that recurring exception handling failures in Java programs are caused by uncaught exceptions that stem from re-mapping sites that re-mapped the caught exception to unchecked types (e.g., `RuntimeException`). Thus, re-mapping violations may pinpoint to potential faults of this type.

7.2.4 Violations in Propagation and Re-Throwing Sites

We observed propagation and re-throwing violations only in the context of Health Watcher. There is no specific fault category proposed by Barbosa et al. and Ebert et al. related to propagation and re-throwing of exceptions. The propagation violations were caused because compartments *Façade* and *View* propagated exceptions that only the *Distribution* compartment is allowed to propagate. Similarly, the rethrow violation occurred because the *Data* compartment rethrows an exception that only the *Persistence* compartment is allowed to re-throw. These violations also resemble the definition of the *"Exception Caught at the Wrong Level"* in the sense that the exception handling dependency relation was implemented in the wrong place of the system.

7.3 Important Considerations

The analyses presented in this section were performed using the *EPL Verifier*. As we discussed in Section 5, 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, re-mapping and re-throwing 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, for these target systems, the use of the *EPL Verifier* did not interfere the results discussed for these target systems.

For the case study performed in the context of the Tomcat target system, all the *throw* statements within the compartments analyzed referred to new instance creation expressions. Therefore, the type-inference algorithm did not

TABLE 6
Comparison of EPL with Related Works

Solutions	Semantics of Rule Types	Supported E.H. Dependency Relations
EPL	Obligation, Permission, Prohibition	Handle, Propagate, Raise, Re-map, Re-throw
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

introduce any imprecision in the analysis presented in this section. Even so, we analyzed the source code of the whole system to identify the cases in which the *EPL Verifier* would have produced imprecise results. From the total of 1,880 *throw* statements in the system, in 1,582 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 one 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, one 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. Thus, 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 it=
    (InvocationTargetException)t;
  throw it.getCause() != null ? it.getCause() :
  it;
}
```

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. Thus, 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.

8 Related Work

The DSL proposed in this work is a means to assure exception handling quality by 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 [27] and Van Ommering et al. [46], these architectural solutions can be divided in three main categories: (i) Reflexion *Model*s, (ii) Relation Conformance Rules and (iii) Component Access *Model*s. Reflexion *Model*s [35] 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 exists 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 [18], [23], [38], [45] specify design rules that express allowed or forbidden relations between architectural elements. Finally, Component Access *Model*s 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 (ADLs) [10]. Among these three categories, our proposed solution has more similarities with those based on relation conformance rules. Table 6 presents a comparison of EPL with related works based on Relation Conformance Rules.

As can be observed in Table 6, in terms of the semantics of the provided rule types, EPL and the solutions proposed by Gurgel et al. [23] and by Terra and Valente [45] 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 dependency relations, EPL is the only solution that supports all the "canonical" dependency relations between exceptions and code elements described in Section 2. The other solutions only support the *Handle*, *Propagate* and *Raise* dependency relations. Thus, when compared to these other solutions, the main contribution of EPL is to provide a wider vocabulary of exception handling dependency relations to specify exception handling policies. Next, we detail other similarities and differences between EPL and the other solutions.

The solution proposed by Sales and Coelho [38] is the most similar to ours. Their solution is also aimed at specifying exception handling policies, although they do not support all the "canonical" exception handling dependency relations. Our solutions mainly differ in the manner we express exception handling policies. While we express our exception handling policies in terms of compartments and their exception handling responsibilities, Sales and Coelho express their exception handling policies in terms of "exceptional contracts". An exceptional contract specifies an intended exception flow, 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. Based on their exceptional contracts, Sales and Coelho generate partial JUnit test cases to stimulate the exceptional behavior of the system. Moreover, their solution requires the intervention of developers to finish the implementation of the partially generated test cases, whereas the *EPL Verifier* requires only the exception handling policy specification and the system source code. In addition, due to limitations of static analysis techniques for exception handling, such as those in the type-inference algorithm implemented by the *EPL Verifier* (Section 5.2), the verification approach adopted by EPL may be more prone to less precise results when compared to the dynamic approach adopted by Sales and Coelho. For this reason, we believe that these two verification approaches may be used in collaboration. 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.

The solutions proposed by Eichberg et al. [18], Gurgel et al. [23] and Terra and Valente [45] specify architectural design rules aimed at detecting and preventing architectural degradation problems. For this reason, these solution focus on expressing architectural design rules in terms of dependency relations originated from source code elements accessing methods and fields, instantiating new class instances, extending classes, implementing interfaces, among others. Not all dependency relations related to exception handling are supported by their solutions. Eichberg et al. considers the *Handle* and *Raise* dependency relations as the generic relation 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, their solutions 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 dependency relations combined with logic operators for conjunction, disjunction and negation. All the other solutions, EPL inclusive, may only express their rules in terms of atomic dependency relations. 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 dependency relations 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 EHMs of programming languages to support the explicit specification of exception handling rules in the source code [9], [42]. Cacho et al. [9] extended the EHM of AspectJ, whereas Silva and Castor [42] extended the EHM 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 EHMs implemented in programming languages. We also designed it to produce readable specifications. The observations of our user-centric study suggest 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 Verifier*s 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.

Previous works from Barbosa et al. [2] and Ebert et al. [17] analyzed the bug history of open source systems to assess and better comprehend faults in the exception handling code. Ebert et al. have also employed a survey with developers to gather their opinions and experiences about faults related to exception handling. Both works proposed a set of categories of exception handling faults. Most of the categories proposed by these works are equivalent. A detailed comparison of the proposed categories is presented by Ebert et al. in their work [17]. As we discussed in

Section 7.2, violations of EPL may pinpoint to potential faults of six different fault categories. These categories are present in the categorizations proposed by Barbosa et al. and by Ebert et al. with equivalent names and definitions. However, not all categories of faults defined by Barbosa et al. and by Ebert et al. can be detected with the help of EPL. Specifications in EPL are meant to contain information at the design level; specific implementation details are beyond the scope of EPL policies. For this reason, categories of exception handling faults related to specific implementation details cannot be detected by means of violations of EPL rules. For example, exception handling faults categorized by Ebert et al. as *"Error in the handler"*, *"Error in the clean up action"*, *"Error in the exception assertion"* and *"Catch block where only a finally would be appropriated"* would require detailed information about how *catch* blocks, *finally* blocks and *if* statements are implemented. Similarly, faults categorized by Barbosa et al. as *"Improper continuation of execution"*, *"Premature termination"*, *"Exceptional loop break"* would also require more detailed information about the implementation. During our case studies, we also observed violations of EPL rules that are not specifically related to any category of exception handling fault, but that are very similar to other fault categories. For example, we observed re-mapping violations caused by re-mapping the caught exception to a generic exception type, which is similar to the fault category *"Uninformative Generic Type Thrown"*. For this reason, we also plan to extended the categorizations of exception handling faults by further investigating if these violations that we observed may actually cause faults in software systems.

Finally, in the exception handling literature there are also a few efforts to support developers in properly designing exception handling of software systems. Litke [30] 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 modules boundaries. Robillard and Murphy [37] proposed a method to design robust Java systems by adapting Litke's method. These methods provide good methodology for specifying exception handling in modules 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 [31] 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 systems' exception handling policies. Thus, we plan to elaborate guidelines to help developers in how to specify their exception handling policies.

9 Final Remarks

In an increasingly software-dependent society, software failures may have severe and negative outcomes. In cases where system failures have severe consequences, important quality attributes are those related to reliability. One of such quality attributes is software robustness. By using exception handling mechanisms, it is hoped that more robust programs are implemented. However, previous studies reported that recurring robustness problems in software systems are often related to problems in the exception handling code [2], [7], [8], [32], [33], [39]. Most of these problems are related to the lack of explicit exception handling policies. In this paper, we presented EPL, a domain-specific language to specify and verify exception handling policies. 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 and across 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 verify if the source code adheres to the specified rules.

With our user-centric observational study, 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 its use, especially in terms of expressiveness and usability. 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 studies revealed that violations of verifiable exception handling policies could help to directly detect faults in the exception handling code. These faults could not be detected with the basic verification performed by reliability-driven EHMs, 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. 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.

With regards to the EPL design, we intentionally designed it not to support the definition of what specific implementation actions exception handlers should do (e.g., retrying, logging, etc.). We believe that this type of specific implementation details should not be included in system-level design models, such as exception handling policies, because specifying very specific implementation details of modules would break the information hiding principle. In fact, in some scenarios, the features provided by EPL may be used in manners that break the information hiding principle. For instance, in some cases, specifying the exceptional behavior of a compartment in terms of the *Raise* dependency type may require too much implementation details; it may be enough to specify its exceptional behavior only in terms of the *Propagate* dependency type. However, in other cases, the same feature may be necessary to specify a given exception handling rule. For example, one of the exception handling rules shown in Section 7.1 for the Tomcat system required the *Raise* dependency type. For this reason, we kept these features in EPL, even if in some cases they may be used to break the information hiding principle. We believe that is up to software architects and designers to achieve the needed balance between the conservation of the encapsulation of code elements and the detailed specification of their exceptional behavior.

Another reason why we did not design EPL to specify which specific exception handling actions should be taken is that specifying this kind of information would make our language too complex, given the high number of possible handling actions that can be implemented. There can even be exception handlers that perform more than one handling action. For instance, an 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 our main design goal, 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.

Finally, recommender systems are being explored as a tool to support developers in implementing exception handling code over the last years [3], [4], [36]. We plan to integrate recommending techniques to the EPL implementation. Our goal is to assist developers in implementing and maintaining policy-compliant exception handling code with recommendations.

ACKNOWLEDGMENTS

Eiji Adachi Barbosa has been funded with support from Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ), grant E-26/100.386/2014.

REFERENCES

[1] E. A. Barbosa and A. Garcia, "Analyzing exceptional interfaces on evolving frameworks," in *Proc. IEEE 5th Latin-American Symp. Dependable Comput. Workshops*, 2011, pp. 17–20.

[2] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing faults in exception handling: A study of open source projects," in *Proc. XXVIII Brazilian Symp. Softw. Eng.*, 2014, pp. 11–20.

[3] E. A. Barbosa, A. Garcia, and M. Mezini. (2012, Jun.). A recommendation system for exception handling code. *Proc. IEEE 5th Int. Workshop Exception Handling*, pp. 52–54 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6226601

[4] E. A. Barbosa, A. Garcia, and M. Mezini. Heuristic strategies for recommendation of exception handling code. *Proc. 26th Brazilian Symp. Softw. Eng.*. Sep. 2012, pp. 171–180. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6337870

[5] J. Bloch, *Effective Java*, series The Java Series. Englewood Cliffs, NJ, USA: Prentice-Hall, 2008.

[6] P. H. S. Brito, R. Lemos, C. M. F. Rubira, and E. Martins. (2009, Apr.). Architecting fault tolerance with exception handling: Verification and validation. *J. Comput. Sci. Technol.* [Online]. *24(2)*, pp. 212–237 [Online]. Available: http://dl.acm.org/citation.cfm?id=1599001.1599006

[7] 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 *Proc. 30th Int. Conf. Softw. Maintenance Evolution*, 2014, pp. 31–40.

[8] N. Cacho, E. A. Barbosa, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, and A. Garcia. (2014). Trading robustness for maintainability: An empirical study of evolving C# programs. *Proc. 36th Int. Conf. Softw. Eng.*, pp. 584–595 [Online]. Available: http://dl.acm.org/citation.cfm?id=2568308

[9] N. Cacho, F. Castor, A. Garcia, and E. Figueiredo, "EJFlow: Taming exceptional control flows in aspect-oriented programming," in *Proc. 7th Int. Conf. Aspect-Oriented Softw. Develop.*, 2008, pp. 72–83.

[10] P. C. Clements. (1996, Mar.). A survey of architecture description languages. *Proc. 8th Int. Workshop Softw. Specification Des.*, p. 16 [Online]. Available: http://dl.acm.org/citation.cfm?id=857204.858261

[11] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa, and C. Lucena. (2008, Jul.). Assessing the impact of aspects on exception flows: An exploratory study. *Proc. Eur. Conf. Object-Oriented Program.*, vol. 5142, pp. 207–234 [Online]. Available: http://dl.acm.org/citation.cfm?id=1428508.1428522

[12] J. W. Creswell and V. L. P. Clark, *Designing and Conducting Mixed Methods Research*. Newbury Park, CA, USA: Sage, 2007.

[13] F. Cristian. (1982, Jun.). Exception handling and software fault tolerance. *IEEE Trans. Comput.* [Online]. *C-31(6)*, pp. 531–540 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1676035

[14] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quart.*, vol. 13, pp. 319–340, 1989.

[15] R. de Lemos and A. Romanovsky. (2001, Mar.). Exception handling in the software lifecycle [Online]. Available: http://kar.kent.ac.uk/13633/

[16] F. Ebert and F. Castor. (2013, Sep.). A study on developers' perceptions about exception handling bugs. *Proc. IEEE Int. Conf. Softw. Maintenance*, pp. 448–451 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6676929

[17] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in Java programs," *J. Syst. Softw.*, vol. 106, pp. 82–101, 2015.

[18] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. (2008, May). Defining and continuous checking of structural program dependencies. *Proc. 13th Int. Conf. Softw. Eng.*, p. 391 [Online]. Available: http://dl.acm.org/citation.cfm?id=1368088.1368142

[19] E. Figueiredo and N. Cacho. (2008). Evolving software product lines with aspects. *Proc. 13th Int. Conf. Softw. Eng.*, p. 261. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4814137

[20] M. Fowler, *Domain-Specific Languages*. Noida, India: Pearson Edu., 2010.

[21] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu. (2001, Nov.). A comparative study of exception handling mechanisms for building dependable object-oriented software. *J. Syst. Softw.*. *59(2)*, pp. 197–222 [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(01)00062-0

[22] J. B. Goodenough. (1975). Exception handling: Issues and a proposed notation. *Commun. ACM* [Online]. *18(12)*, p. 683. Available: http://portal.acm.org/citation.cfm?id=361230

[23] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke. (2014, Apr.). Blending and reusing rules for architectural degradation prevention [Online]. *Proc. 13th Int. Conf. Modularity*, pp. 61–72. Available: http://dl.acm.org/citation.cfm?id=2577080.2577087

[24] IEEE. (1990, Dec.). IEEE standard glossary of software engineering terminology. pp. 1–84 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=159342

[25] B. Jackobus, A. Garcia, E. A. Barbosa, and C. J. Lucena, "Contrasting exception handling code across languages: An analysis of 50 open source projects," in *Proc. 26th Int. Symp. Softw. Rel. Eng.*, 2015, pp. 189–200.

[26] J. Kienzle. (2008, Nov.). On exceptions and the software development life cycle. *Proc. 4th Int. Workshop Exception Handling*, pp. 32–38 [Online]. Available: http://dl.acm.org/citation.cfm?id=1454268.1454273

[27] J. Knodel and D. Popescu. (2007, Jan.). A comparison of static architecture compliance checking approaches. *Proc. Working IEEE/IFIP Conf. Softw. Archit.*, pp. 12–12 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4077029

[28] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. Staa, and C. Lucena. (2006, Sep.). Quantifying the effects of aspect-oriented programming: A maintenance study. *Proc. 22nd IEEE Int. Conf. Softw. Maintenance*, pp. 223–233 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4021341

[29] B. Liskov and A. Snyder. (1979, Nov.). Exception handling in CLU. *IEEE Trans. Softw. Eng.* [Online]. *SE-5(6)*, pp. 546–558 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1702672

[30] J. D. Litke. (1990, Dec.). A systematic approach for implementing fault tolerant software designs in Ada. *Proc. Conf. TRI-ADA*, pp. 403–408 [Online]. Available: http://dl.acm.org/citation.cfm?id=255471.255565

[31] D. Malayeri and J. Aldrich, "Practical exception specifications," in *Advanced Topics in Exception Handling Techniques*. New York, NY, USA: Springer, 2006, pp. 200–220 [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.1502

[32] C. Marinescu. (2011, Sep.). Are the classes that use exceptions defect prone? *Proc. 12th Int. Workshop 7th Annu. ERCIM Workshop Principles Softw. Evolution Softw. Evolution*, p. 56 [Online]. Available: http://dl.acm.org/citation.cfm?id=2024445.2024456

[33] C. Marinescu. (2013, Sep.). Should we beware the exceptions? An empirical study on the eclipse project. *Proc. IEEE 15th Int. Symp. Symbolic Numeric Algorithms Sci. Comput.*, pp. 250–257 [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6821157

[34] B. Meyer, *Eiffel: The Language*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[35] G. C. Murphy, D. Notkin, and K. Sullivan. (1995, Oct.). Software reflexion models. *ACM SIGSOFT Softw. Eng. Notes* [Online]. *20(4)*, pp. 18–28 Available: http://dl.acm.org/citation.cfm?id=222132.222136

[36] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *Proc. 14th Int. Working Conf. Source Code Anal. Manipulation*, 2014, pp. 285–294.

[37] M. P. Robillard and G. C. Murphy. (2000). Designing robust Java programs with exceptions. *Proc. 8th ACM SIGSOFT Int. Symp. Found. Softw. Eng.: 21st Century Appl.*, pp. 2–10 [Online]. Available: http://portal.acm.org/citation.cfm?id=357474.355046

[38] R. Sales and R. J. Coelho, "Preserving the exception handling design rules in software product line context: A practical approach," in *Proc. IEEE 5th Latin-American Symp. Dependable Comput. Workshops*. 2011, pp. 9–16.

[39] P. Sawadpong, E. B. Allen, and B. J. Williams. (2012, Oct.). Exception handling defects: An empirical study. *Proc. IEEE 14th Int. Symp. High-Assurance Syst. Eng.*, pp. 90–97 [Online]. Available: http://www.computer.org/csdl/proceedings/hase/2012/4912/00/4912a090-abs.html

[40] H. Shah, C. Gorg, and M. Harrold. (2010, Mar.). Understanding exception handling: viewpoints of novices and experts. *IEEE Trans. Softw. Eng.*, *36(2)*, pp. 150–161 [Online]. Available: http://dl.acm.org/citation.cfm?id=1803947.1804209

[41] H. Shah and M. J. Harrold. (2009). Exception handling negligence due to intra-individual goal conflicts. *Proc. ICSE Workshop Cooperative Human Aspects Softw. Eng.*, pp. 80–83 [Online]. Available: http://portal.acm.org/citation.cfm?id=1572220

[42] T. B. L. Silva and F. Castor, "New exception interfaces for Java-like languages," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, Mar. 2013, pp. 1661–1666.

[43] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 849–871, Sep. 2000.

[44] S. Soares, E. Laureano, and P. Borba. (2002, Nov.). Implementing distribution and persistence aspects with aspectJ. *ACM SIGPLAN Notices* [Online]. *37(11)*, p. 174 Available: http://dl.acm.org/citation.cfm?id=583854.582437

[45] R. Terra and M. T. Valente. (2009, Aug.). A dependency constraint language to manage object-oriented software architectures. *Software—Practice Experience* [Online]. *39(12)*, pp. 1073–1094 Available: http://dl.acm.org/citation.cfm?id=1573951.1573954

[46] R. van Ommering, R. Krikhaar, and L. Feijs. (2001, Apr.). Languages for formalizing, visualizing and verifying software architectures. *Comput. Languages* [Online]. *27(1–3)*, pp. 3–18 Available: http://dl.acm.org/citation.cfm?id=2245755.2245982

Eiji Adachi Barbosa received the MSc degree in informatics from the Pontifical Catholic University of Rio de Janeiro. He is currently working toward the PhD degree at the Informatics Department, Pontifical Catholic University of Rio de Janeiro. His research focuses on exception handling and recommender systems for software engineering. He is a member of the IEEE.

Alessandro Garcia received the PhD degree in informatics from the Pontifical Catholic University of Rio de Janeiro. He is an associate professor at the Informatics Department, Pontifical Catholic University of Rio de Janeiro. His research focuses on software modularity, software metrics, exception handling and empirical software engineering. He is a member of the IEEE.

Martin P. Robillard received the PhD degree in computer science from the University of British Columbia. He is an associate professor at the School of Computer Science, McGill University. His research focuses on software archive mining, API usability, and recommender systems. He is a member of the IEEE.

Benjamin Jakobus received the MSc degree in advanced computing from the Imperial College London. He is currently working toward the PhD degree at the Informatics Department, Pontifical Catholic University of Rio de Janeiro. His research focuses on programming languages and domain-specific languages. He is a member of the IEEE.