Improving Exception Handling with Recommendations

Eiji Adachi Barbosa
OPUS Research Group, Informatics Department, PUC-Rio
Rua Marquês de São Vicente, 255, Gávea, 22453-900
Rio de Janeiro-RJ, Brazil
ebarbosa@inf.puc-rio.br

ABSTRACT

Exception handling mechanisms are the most common model used to design and implement robust software systems. Despite their wide adoption in mainstream programming languages, empirical evidence suggests that developers are still not properly using these mechanisms to achieve better software robustness. Without adequate support, developers struggle to decide the proper manner in which they should handle their exceptions, i.e., the place where the exception should be caught and the handling actions that should be implemented. As a consequence, they tend to ignore exceptions by implementing empty handlers or leaving them unhandled, which may ultimately lead to the introduction of faults in the source code. In this context, this PhD research aims at investigating means to improve the quality of exception handling in software projects. To achieve this goal, we propose a recommender system able to support developers in implementing exception handling.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

General Terms

Design, Reliability.

Keywords

Exception handling, recommender system.

1. INTRODUCTION

Exception handling mechanisms [4] (EHMs, for short) are intended to ease the design and implementation of robust software. EHMs are typically provided as built-in features in most mainstream programming languages, such as Java, C#, C++, among others. However, despite their wide acceptance in programming languages, developers are still not properly using EHMs to achieve software robustness.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

Poor quality of exception handling. Independently of programming language, the exception handling code observed in industry systems is poor [2,7]. From 40% to 70% of the exception handlers (catch blocks) implemented in industry systems perform overly simplistic actions, such as only printing the stack trace, returning null or even doing nothing (with an empty catch block) [2]. Developers seem to implement overly simplistic handlers just to silence automatic reliability checks, such as those performed by the Java compiler. When there is no automatic reliability checks, as in the C# and C++ languages, they simply ignore the occurrence of exceptions, leaving them unhandled, and even implement empty catch blocks [2]. Overly simplistic handlers and unhandled exceptions are a common source of failures in software systems [2, 7]. In fact, the exception handling code is more error-prone than the normal code and the defect density of the exception handling code is almost three times higher than the defect density of the normal code [7]. The truth is that implementing exception handling code without proper support is still a daunting task for most developers. Next, we introduce a typical scenario in which exceptions occur to motivate this PhD research.

Motivating example. As a motivating example consider the following scenario extracted from the MobileMedia [3] (MM, for short), which is a mobile application responsible for managing albums of photos, videos and audio files in mobile devices. The MM is implemented in the Java Platform Micro Edition – Java ME – and its architecture adheres to the Model-View-Controller pattern. Figure 1 illustrates the scenario in which an exception occurs during the implementation of the functionality that removes a photo from a given album. The figure depicts a simplified code snippet that implements part of the call chain responsible for deleting a photo from a given album in the MM.

In the code snippet depicted in Figure 1, an event in the `View` is coped by the `PhotoScreen.handleEvent` method. The `PhotoScreen.handleEvent` method invokes the `Controller.performAction` method to trigger the corresponding actions. Since in this scenario the coped event regards to a photo removal, the `Photo.remove` method is invoked. The `Photo.remove` method invokes the `PhotoAccessor.delete` method to actually delete the selected photo from the given album and the `PhotoScreen.update` method to update the screen with a successful message after the deletion. The `PhotoAccessor.delete` method deletes the selected photo by accessing the device storage system through the `RecordStore` API. When the developer invokes the `RecordStore.deleteRecord` method, he realizes that this method may

Figure 1: Implementing exception handling

throw an instance of `RecordStoreException` during its execution. At this point, the developer must decide the manner in which they will handle it. Next, we show the decision making process involved to properly handle the exception of the motivating example.

Deciding proper exception handling. Developers are supposed to use EHMs constructs to implement software systems able to continually provide their expected functionalities even in the presence of exceptions. In the example depicted in Figure 1, an instance of `RecordStoreException` represents an error that occurs when accessing the device storage system. Therefore, if this exception occurs, it is not possible to remove the selected photo from a given album, and there is nothing else that could be done to workaround the error. Hence, the application should warn its user about this error. To achieve this requirement, it is necessary to decide where and how the exception should be handled. Thus, it is necessary to propagate the exception from the scope where it occurred (within the `Model` component) up to the `Controller` component 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` component to the scope of `Controller` component would break information hiding principles, i.e., the `Controller` component would be aware of implementation details of the `Model` component. To avoid this, exceptions that cross the boundaries of components should be remapped to more abstract exception types in order to preserve information hiding. In the example, when `RecordStoreException` crosses the boundaries of the `Model` and `Controller` components, it could be remapped to an exception type called `PersistenceException`, for instance. Thus, the `Model` component gives information about the cause of the problem, but does not expose implementation details. Moreover, the `Controller` component 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).

2. PROBLEM STATEMENT

Ignoring exception handling. Differently from the approach shown in Section 1, when it comes to handling exceptions in practice, the most common approach adopted by developers is ignoring exceptions [12]. Developers ignore exception handling in the first versions of a system and only try to improve it in further versions. However, they usually struggle to improve exception handling code in future versions due to their initial poor decisions regarding how to use exceptions. In particular, the excessive use of generic exceptions, the high number of unhandled exceptions and ineffective handlers (e.g. empty catch blocks) in early versions of a system hinder improvements of exception handling in future versions. Moreover, although ignoring exceptions may seem a harmless approach to many developers, it may actually introduce faults in the source code [7]. In the context of the example depicted in Figure 1, one developer could simply ignore the occurrence of the exception and leave it unhandled. Another developer could simply catch the exception in the scope of the `PhotoAcessor.delete` method and ignore its handling by implementing an empty catch block. In both cases, ignoring the occurrence of `RecordStoreException` introduces a fault in the application. In the first case, there is no handler for the exception, so if the exception occurs, the application is terminated. In the second case, the photo is not removed from the device due to the exception, the user is not aware of the problem, but the application continues to execute normally, updating the screen with a successful message (invocating the `PhotoScreen.update` method in the code snippet depicted in Figure 1). Although one may argue that these harmful scenarios of improper exception handling seem unusual, or overly trivial, they are actually commonly observed in practice [2, 7, 8].

Inadequate support for exception handling. Without support, most developers fail in implementing proper exception handling [2, 7, 8]. In worst cases, they even introduce faults in the source code [7]. For this reason, over the last years solutions aimed at supporting developers handling exceptions have been proposed [6]. Most of these tools are based on static analysis techniques that compute the possible propagation paths of exceptions in a program. These tools are mainly used to aid developers to comprehend the structure of their programs in the presence of exceptions and how exceptions themselves behave in order to perform maintenance tasks. The propagation paths of exceptions are also used to detect unhandled exceptions in languages that do not provide EHM with built-in static reliability checks, such as C++ and C#. However, these tools are not useful during the implementation of exception handling. For instance, in the implementation scenario depicted in Figure 1, in the moment when an exception occurrence is identified, these tools are not able to assist developers in deciding where and how they should handle the exception. Without this kind of support, developers are forced to decide on their own and usually implement overly simplistic handlers or simply ignore the exceptions. Therefore, a better support for developers implementing exception handling is needed. More recently, recommender systems have emerged as a promising means to support software engineering activities [5]. Although many recommender systems for supporting software engineering activities have already been proposed, our previous recommender is the only one tailored to aid developers in implementing exception handling [1]. This recommender system assumes that developers know where they should handle exceptions, and they only need support for implementing exception handlers (catch blocks). But as

previously discussed, most developers are failing in deciding where and how they should handle exceptions. Therefore, the assumption of the only recommender system for exception handling does not always hold. In other words, current support for developers implementing exception handling is still limited. In this context, the main problem tackled by this PhD research is the lack of adequate support for developers in the process of handling exceptions; in particular, in the process of deciding where and how they should handle their exceptions.

Absence of explicit exception handling policies. Part of the limitations of current supporting tools for exception handling presented in the previous paragraphs stem from the absence of an explicit exception handling policy. An exception handling policy refers to software designers' intent regarding how and where exceptions should be used and handled in the context of a software project. Specifications of exception handling policies are not typically part of most software development processes. In general, they are not formally specified and are, at best, only partially defined as scattered comments in the source code. Without an explicit specification of the intended use of exceptions, supporting tools cannot reason about whether the implemented exception handling code adheres to the intended exception handling policy or not. Thus, these tools cannot support developers in implementing exception handling code that is adherent to designers' intent, nor automatically detect in the source code deviations from designers' intent. Therefore, these tools cannot provide more sophisticated support for developers implementing exception handling. In fact, there is a lack in the literature in terms of means to specify exception handling policies. Previous solutions [3] extended programming languages with new constructs to allow developers specifying exception handling properties, such as the place where exceptions may be raised and handled. These properties are part of an exception handling policy, but are not sufficient to specify some exception handling policies. For instance, policies regarding remapping of exceptions, as exemplified in Section 1, cannot be explicitly specified with the constructs provided by these solutions. Therefore, the second problem tackled by this PhD research is the lack of means to explicitly specify exception handling policies in software projects.

3. RESEARCH QUESTIONS

This PhD research aims at investigating means to aid developers to implement exception handling code that actually improves software robustness. As shown in Section 2, leaving the implementation of proper exception handling to future versions of a system may have adverse consequences to software quality. Therefore, our main goal is to support developers in the implementation of exception handling since the first versions of a system. We plan to achieve this goal by building a recommender system able to support developers in the decision making process of the manner in which developers should handle exceptions, i.e., deciding where and how developers should handle their exceptions. With this recommender system we aim at bridging the gap of current supporting tools, which are mainly focused on the maintenance of exception handling code, nor in its implementation. Also, we aim at raising the level of current recommender system for exception handling by relying on explicit exception handling policies specifications to perform more sophisticated recommendations. Therefore, it is also a goal of this PhD research to propose a means to specify exception handling policies. To achieve these goals, we aim at investigating the following research questions:

RQ1: How to explicitly specify exception handling policies?

RQ2: How to assist developers in the decision of where and how they should handle their exceptions?

The next section details the activities performed to achieve the research goals and answer the research questions.

4. RESEARCH ACTIVITIES

Defining a specification language. Due to the lack of means to specify exception handling policies (Section 2), we defined a declarative language independent of programming language to specify exception handling properties. The main design goals of the language were conciseness and readability. Thus, we defined a specification language with few constructs and whose specifications were similar to specifications in natural language. In this manner, exception handling policies are specified by means of a limited vocabulary of high level exception handling terms. The core concepts of the proposed language are: compartments [6] and rules. The following example specifies the exception handling policy followed in the "Expected exception handling" example presented in Section 1 and helps to introduce the core concepts of the language.

```
1. define pre.model.* as compartment MODEL;
2. define pre.control.* as compartment CONTROLLER;
3. define rule MODEL must remap RecordStoreException to PersistenceException;
4. define rule CONTROLLER must handle PersistenceException;
```

In the first two lines of the code snippet depicted above, we specify that the code elements whose fully qualified names start with the prefixes `pre.model` and `pre.controller` are defined as the compartments `MODEL` and `CONTROLLER`, respectively A compartment is a set of code elements in the source code that may throw exceptions. Compartments can be seen as logical barriers where exception handling rules are applied. In the following lines of the previous example two rules are specified. A rule creates one type of dependency between compartments and exception types. In the first rule, for instance, the rule specifies that the `MODEL` compartment is forced to remap any occurrence of the `RecordStoreException` to the `PersistenceException`. The second rule specifies that the `CONTROLLER` compartment is forced to handle any occurrence of a `PersistenceException`. In other words, these rules specify that instances of `RecordStoreException` must not leave the boundaries of the `MODEL` compartment. Instead, these exceptions must be remapped to `PersistenceException` which, in turn, are handled by the `CONTROLLER` compartment.

We are currently in the evaluation phase of the proposed language. In order to assess the ability of the language to express exception handling policies with its small vocabulary, we performed a case study in the context of the MobileMedia. We performed this first case study using the MobileMedia as our subject system because we have contact with its original designers, therefore we could question them about the intended exception handling policy and ask them to check the specifications produced. The initial results of this case study show that the constructs provided

by the proposed specification language are enough to specify a wide variety of exception handling rules. We plan to perform other case studies in the context of other systems to further assess the language.

As previously stated, the design goals of the language were conciseness and readability. These are two design goals commonly followed by domain-specific languages to decrease the learning cost and increase the language acceptance. In order to assess these assumptions, we are preparing an observational study followed by a semi-structured interview with software designers. In this study we will ask software designers to use the language to specify one of their systems in order to identify possible barriers that may hinder the learning and acceptance of the proposed specification language. The results of this study will complement the results gathered in the case studies and will allow us to identify opportunities to make minor improvements in the language. These results will also allow us to assess the need for auxiliary procedures to systematize the use of the language.

Building a recommender system. We plan to enrich current recommending techniques with global information about exceptions specified in exception handling policies. By having at disposal an explicit specification of an exception handling policy, a better support for developers implementing exception handling code can be provided. For example, when a developer invokes the `RecordStore.deleteRecord` method and he realizes that this method raises the `RecordStoreException`, as in the example depicted in Figure 1, the recommender system could be triggered to recommend the following steps:

(i) Propagate `RecordStoreException` to `Photo.remove` method from the scope of `PhotoAcessor.delete`;

(ii) Propagate `RecordStoreException` to `Controller.performAction` method from the scope of `Photo.remove`;

(iii) Catch and handle `RecordStoreException` within the scope of `Controller.performAction`.

To build the envisioned recommender system it is necessary to implement a tool able to verify if the source code adheres to an exception handling policy. Although the proposed language for specifying exception handling policies is independent of programming language, the verification tool has to be dependent of programming language, since it involves analyzing properties of the source code. Therefore, we had to opt which programming language to support in the context of this PhD research. We opted to support the C# language because its EHM contains characteristics common to most mainstream programming languages: no support for declaration of exceptions in methods signatures, automatic propagation of exceptions, dynamic reliability checks and no static reliability check. In fact, among the Top 10 [1] most popular programming languages , only Java, which supports declaration of exceptions in methods signatures and performs static reliability checks, and C, which does not provide a built-in EHM, do not share these characteristics.

To assess the verification tool in terms of precision and recall, we plan to perform case studies in the context of systems that we can have access to its designers to use them as oracles. In this manner, we will be able to specify the systems' intended exception handling policy and validate the detected deviations in the source code, as well as deviations not detected by the tool.

[1]http://www.tiobe.com/index.php/content/paperinfo/tpci

Along with the case studies, we plan to perform interviews with the designers in order to gather deeper knowledge about common problems regarding exception handling and how exception handling code is maintained. This will provide us valuable insights that will be further used in the definition of recommending heuristics. These recommending heuristics will be implemented as the recommendation engine of our recommender system. Our goal is to implement the recommender system integrated to an IDE so it can be integrated more easily to developers workflow. Finally, we plan to perform controlled experiments in order to assess if the use of the recommender system can actually improve the quality of the exception handling code in terms of the number of faults in the exception handling code. We also plan to perform qualitative studies, such as observational studies and interviews, to identify possible improvements in the recommender system, specially in terms of how the recommendations are triggered and presented to users.

5. EXPECTED CONTRIBUTION

The expected contributions of this PhD research are: (i) a declarative language tailored to specify exception handling policies; (ii) an automatic verification tool to support the adherence check of an exception handling policy; (iii) lessons learned on how information from an exception handling policy can be used to enrich current recommending techniques; (iv) a set of recommender heuristics that supports developers in the implementation and maintenance of exception handling code; and (v) a recommender system that implements the proposed recommender heuristics.

6. REFERENCES

[1] Barbosa, E. A., Garcia, A., and Mezini, M. Heuristic Strategies for Recommendation of Exception Handling Code. In *Proc. of the 26th SBES* (Sept. 2012), IEEE, pp. 171–180.

[2] Cabral, B., and Marques, P. *Exception Handling: A Field Study in Java and .NET*, vol. 4609 of *LNCS*. Springer Berlin Heidelberg, 2007.

[3] Cacho, N., Castor, F., Garcia, A., and Figueiredo, E. EJFlow : Taming Exceptional Control Flows in Aspect-Oriented Programming. In *Proc. of the 7th AOSD* (2008), pp. 72–83.

[4] Goodenough, J. B. Exception handling:Âăissues and a proposed notation. *Communications of the ACM 18*, 12 (1975), 683.

[5] Robillard, M., Walker, R., and Zimmermann, T. Recommendation Systems for Software Engineering. *IEEE Software 27*, 4 (July 2010), 80–86.

[6] Robillard, M. P., and Murphy, G. C. Designing robust Java programs with exceptions. In *Proc. of the 8th ACM SIGSOFT FSE* (2000), pp. 2 – 10.

[7] Sawadpong, P., Allen, E. B., and Williams, B. J. Exception Handling Defects: An Empirical Study. In *2012 IEEE 14th HASE* (Oct. 2012), IEEE, pp. 90–97.

[8] Shah, H., Gorg, C., and Harrold, M. Understanding Exception Handling: Viewpoints of Novices and Experts. *IEEE TSE 36*, 2 (Mar. 2010), 150–161.