A Recommendation System for Exception Handling Code

Eiji Adachi Barbosa, Alessandro Garcia  
OPUS Research Group  
PUC-Rio, Rio de Janeiro, Brazil  
{ebarbosa, afgarcia}@inf.puc-rio.br

Mira Mezini  
Software Technology Group  
Technische Universität, Darmstadt, Germany  
{mezini}@cs.tu-darmstadt.de

Abstract—Even though exception handling mechanisms are part of most mainstream programming languages, software developers still struggle to implement proper exception handling code. In particular, they fail in implementing effective handler actions. This position paper discusses our ongoing work on implementing and assessing a recommendation system for recommending code fragments implementing exception handling code. These fragments are not meant to be reused as-is. Instead, they are meant to be used by the developers as examples of how to possibly handle their exceptions. The goal of the proposed recommendation system is to assist the learning process of software developers by providing concrete examples of exception handling code.

Keywords—Recommendation System; Exception Handling.

I. INTRODUCTION

Although the importance of exception handling in software development is advocated for a long time [1, 6], implementing the exceptional behavior of a software system is still considered a secondary task by many developers. Not surprisingly, the lack of proper handling actions in software systems is often very common. Previous studies [2] have shown that 40% to 72% of the total distribution of handling actions in common software applications is composed by empty handlers or only-logging handlers. In worst case scenarios, this lack of proper handling actions may even lead to recurring faults [3] and performance issues [3].

To mitigate these problems, considerable efforts have already been made to develop tools that aid developers primarily in detecting poorly implemented handlers [4, 5]. But only detecting them is not enough. The real problem with the poor quality of exception handling code in software systems seems to be much deeper, since most developers struggle and are unwilling to produce any exception handling code. None of the current tools actually aid developers to implement their own handlers from scratch, or improve poor handlers previously implemented.

In this position paper we present our ongoing work on developing and assessing a recommendation system for exception handling code. Our recommendation system recommends a list of code fragments implementing exception handling ranked by relevance. Our aim is to develop a tool that supports developers’ learning by providing code examples. These examples are extracted from open-source projects. We believe that we can leverage on the collective knowledge produced by the open source community to help developers to produce better exception handling code. Optimistically, we expect that by exposing developers to alternative ways of handling exceptions they become more aware of exception handling policies and implement their handlers more properly. We have already performed some preliminary assessments on our recommendation system and the results are encouraging.

II. BASIC ARCHITECTURE

The recommendation system for exception handling code relies on three basic components: Recommendation Engine, Extractor and Repository Manager. The Extractor component is responsible for extracting from application projects the code fragments that are stored in the repository of examples. It is also responsible for extracting structural information, called structural facts, from a given code fragment. The Repository Manager is responsible for storing and indexing the code fragments extracted by the Extractor component. The Recommendation Engine component is responsible for searching, retrieving and ranking code examples stored in the repository of examples based on structural facts extracted from the code fragment the developer is working on.

The components interact as follows in order to recommend a list of handler fragments. The Extractor component first extracts a set of structural facts from the code fragment the developer is working on. Based on these structural facts, the Recommendation Engine builds and executes through the Repository Manager a query in the repository of examples, retrieving the examples that satisfy the query. The Recommendation Engine ranks the examples accordingly to their relevance regarding the query used. The best-ranked candidates are then recommended to the developer.

III. HEURISTIC STRATEGIES

Our recommendation engine relies on three heuristic search strategies: Heuristic of Exception Type (Section III.A), Heuristic of Methods Called (Section III.B) and Heuristic of Variable Types (Section III.C). The heuristics search for code fragments similar to the code fragment under development. Their premise is that searching for similar code fragments will increase the chance of recommending code examples realizing relevant exception handler actions.

The heuristics define the similarity degree of two code fragments based on their structural facts. Each heuristic performs its search based on different structural information, such as the type of the handled exception, the methods
invoked and the types of the variables used. In this first version of our recommendation system we decided to use basic structural information. The goal was to assess if simple heuristic strategies are able to recommend relevant information to developers of exception handling code. Our preliminary results indicate that the proposed heuristics recommend relevant code fragments, despite the triviality of the structural information being exploited. Even then, in future versions of the recommendation system, we plan to consider other structural information and check if they actually improve the precision of our heuristics.

A. Heuristic of Exception Type

Exceptions are typically structured in hierarchy trees in programming languages. Exception types that represent semantically-related exceptional conditions are (ideally) grouped in the same sub-tree. The Heuristic of Exception Type assumes that there is significant code similarity between handlers catching exception types that are closely declared in the hierarchy tree.

So given a code fragment $C_1$ that implements a handler that declares an exception of type $T_1$. The Heuristic of Exception Type searches for code fragments $C_2$ that implement handlers declaring exceptions of type $T_2$, in which $T_2$ has the same type of $T_1$, or $T_2$ is a direct super-type of $T_1$.

B. Heuristic of Methods Called

Properly handling an exception does not rely solely on its type, but also on the context of the method where the exception is being handled. In order to represent the context of a given code fragment, the Heuristic of Methods Called assumes that code fragments that call the same methods have more similar contexts. Therefore, they have bigger chances to implement more similar handlers.

So given a code fragment $C_1$ that calls methods $m_1, m_2, \ldots, m_n$, in which one of the calls $m_{i+1,2}$ throws an exception of type $T_1$, and the developer wants to handle this exception. The Heuristic of Methods Called searches for code fragments $C_2$ that must call $m_1$. The more common calls $C_1$ and $C_2$ share, the more relevant $C_2$ is considered.

C. Heuristic of Variable Types

As an alternative way of representing a method’s context, the Heuristic of Variable Types assumes that methods using variables of same types have more similar context. Therefore, they are more likely to implement more similar handlers.

Given a code fragment $C_1$, consider $V_1$: the set of variables used by $C_1$ and $TV_1$: the set of types of the variables in $V_1$. The Heuristic of Variable Types searches for code fragments $C_2$ that use at least one variable whose type is in $TV_1$. The more common variable types $C_1$ and $C_2$ share, the more relevant $C_2$ is considered.

D. Heuristics Realization

The structural facts required by each heuristic are represented as terms. Each term is a pair $<$Field:Value$>$, where Field represents the name of the structural fact and Value represents the actual value of the fact. Each heuristic defines its set of structural facts with different Field’s. The Heuristic of Exception Type defines its structural facts with the Field ‘handles’, whereas the Heuristic of Methods Called defines with the Field ‘calls’ and the Heuristic of Variable Types defines with the Field ‘uses’.

The recommendation engine composes all terms of all three heuristics in a single disjunctive Boolean expression, which is used to query the repository of examples. In this manner, we are able to submit a single query combining all three heuristics to the repository. For instance, given the set of structural facts $handles = \{IOException\}$, $calls = \{open, read\}$, $uses = \{File, InputStream\}$, the query $Q$ built by the recommendation engine would be as: $Q = handles: IOException OR calls: open OR calls: read OR uses: File OR uses: InputStream$.

IV. Preliminary Results

In our first experiment, we assessed the ability of the heuristic strategies in finding similar code fragments. To do so, we performed the following experiment: for each code example stored in the repository we extracted its structural facts and used them to search for candidates in the repository, recording in which position the original example was recommended. From the 7919 examples stored in the repository, 5535 examples were recommended in the first position (70%) and 6710 were recommended amongst the first ten recommendations (85%).

The code example in Figure 1 depicts why most of the other 1209 examples were not recommended within the first ten recommendations. The method readFileContent depicted in Figure 1 has a limited set of structural facts. Its structural facts are: $\{uses: String, uses: BufferedReader, calls: read, calls: readLine, handles: IOException\}$. As the heuristics base their searches in structural facts extracted from code fragments, a limited set of facts does not provide sufficient information to identify structural similarities among code fragments.

```java
public String readFileContent(){
    String buffer = "";
    BufferedReader bReader = new BufferedReader(...);
    try {
        while(bReader.ready()){
            buffer += bReader.readLine();
        }
    } catch (IOException e) {
    }
    return buffer;
}
```

Figure 1 Code fragment with limited set of structural facts
Additionally, some of these structural facts are exceedingly ordinary. The structural fact uses: String, for instance, is owned by almost 60% of the examples stored in the repository. The heuristics consider two code fragments similar based on the facts they have in common. So when a code fragment has exceedingly ordinary facts, the heuristics find in the repository many different code fragments that also have these facts. In this manner, the combination of a limited set of structural facts and the exceedingly ordinary nature of some of these facts decreases the precision of the heuristics.

In our second experiment, we assessed the relevance of the recommendations provided by our system. To determine whether or not the recommendations had relevant information we first built an oracle. To do so, we manually collected from wiki, FAQs, APIs documentation, forums and mailing lists a set of exception handling guidelines for Eclipse-based applications. These guidelines are defined in terms of classes and methods that might be used to perform Eclipse-specific exception handling tasks, such as instantiating Eclipse-specific exceptions, or instantiating error status objects used by Eclipse to provide details of failures, or invoking proper error status handlers, amongst others.

Next, we randomly selected 20 code fragments from two Eclipse-based applications that were not stored in the repository of examples and run our heuristics on them. For each code fragment, we registered how many examples recommended amongst the first ten in the recommendation list contained at least one of the guidelines defined in our oracle. For all of the 20 code fragments used in the second experiment there were recommendations containing at least one of the guidelines defined in our oracle. From the 20 code fragments used, 12 fragments received recommendations containing relevant information in the first recommendation. There was even one fragment in which we realized that five different recommendations invoked the same method within the catch block. Later, after analyzing Eclipse API documentation, we discovered that this was another guideline for handling exceptions not previously compiled in our oracle. This scenario highlights the concretization of the main feature and motivating idea of our recommendation system: the ability to provide information to the application developer in order to aid him in the process of learning how to handle exceptions in the context of enclosing frameworks being used.

V. Final Remarks

Much has been said about the importance of exception handling in software development [6]. But the fact is that although exception handling mechanisms exist in programming languages since the late 1970s, implementing the exceptional behavior of a system is still a daunting task for most developers. The truth is that it is no longer acceptable to have so many poorly structured handlers in real software applications nowadays.

We believe that we can improve exception handling quality by aiding developers in the learning process. To put that forth, we implemented a first prototype of a recommendation system that recommends a list of code fragments implementing exception handling ordered by relevance. We recommend code fragments that share structural similarities with the code under development in order to increase the chances of recommending more relevant examples. To do so, we developed heuristic strategies that select and rank code examples from a repository based on structural information extracted from the source code under development. Preliminary assessments of our recommendation system showed encouraging results. Our recommendation engine relies solely in the content of the structural facts extracted from the fragment the developer is working on. We plan to explore concepts of collaboration filters, like explicit feedback from users, to refine our ranking mechanism. Moreover, the assessments performed up to now were focused on Eclipse-based applications. We plan to perform broader assessments, incorporating applications of different domains. We also plan to perform controlled experiments with real developers to validate the usefulness of the recommendations provided by our system.

REFERENCES