Heuristic Strategies for Recommendation of 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—Software developers have difficulties in implementing exception handling code in their systems. In particular, they fail in implementing appropriate handling actions. Not surprisingly, recurring failures and performance issues are often related to the poor exception handling actions implemented in real software systems. In this paper we propose, implement and evaluate a set of three heuristic strategies used to recommend exception handling code in programming tasks. Given a method in which the developer wants to handle an exception, the heuristics recommend a list of code fragments implementing exception handling. The heuristics’ goal is to accurately find code fragments implementing exception handling and recommend these fragments according to the context of developers’ implementation task. Hence, the proposed heuristics may assist developers in the process of discovering exception handling actions relevant to their context by providing concrete examples. We believe that a recommender system based on the proposed heuristics may be used in the future as a tool for aiding exception handling implementation.

Keywords - exception handling; heuristic strategies; code example recommendation

I. INTRODUCTION

Robust software systems must deliver their correct services even in the presence of exceptions [1]. An exception is an event that occurs during the normal execution of a software system, indicating that the system internal state is inconsistent. The occurrence of an exception requires the deviation of the normal control flow of a program in order to execute handling actions. Exception handling mechanisms [2] are the most common models used in mainstream programming languages to detect the occurrence of exceptions and to structure handling actions.

Exception handling mechanisms are meant to improve software robustness. However, the current state of practice on exception handling implementation in real software systems is worrying. Previous studies [3] show that from 40% to 72% of the handling actions implemented in real software systems is overly simplified or ineffective. They only log error messages, print the exception stack trace or even do nothing in some cases. There are very few attempts in implementing actions to recover the consistency of the system internal state. The overly simplicity or ineffectiveness of the corrective actions implemented in these software systems is not just a matter of aesthetics of the source code. Recurring failures [4], performance issues [5] and architectural violations [6] are related to the poor implementation of exception handling actions.

When it comes to implementing exception handling in their systems, software developers often adopt an ignore-for-now approach [7]. It is assumed that they postpone the correct implementation of exception handling for future versions of their systems [7]. But this is rarely the case. In our previous studies [8, 9], we have assessed the quality of exception handling code during software evolution. It was very common to observe in these studies overly simplistic and ineffective handlers in the early releases of a system. However, it was very rare the cases during software evolution when developers actually tried to improved the quality of handling actions previously implemented. It was also observed that even after years of software evolution, developers still implement overly simplistic and ineffective handling actions in the same project. It seems that software developers struggle and are unwilling in implementing it. In particular, they seem to have difficulties in defining and implementing effective handling actions. And when the handling actions are not properly chosen, the whole exception handling mechanism loses its utility, becoming a burden that must be carried by the developer.

To mitigate these problems, considerable efforts have already been made to aid developers in comprehending and maintaining exception handling code [10, 11, 12]. All these proposed techniques might be viewed as a series of refinements based on the same idea: exploiting static analysis techniques that generates data to a front-end that shows these data textually or graphically. These techniques mainly fail because they produce vast amounts of information that developers often do not understand, or because the information produced is not necessarily what developers want. These techniques assume that the exception handling code is well structured upfront in software projects. They also assume that only minor problems on exception handling implementation must be detected and fixed in later system versions. However, this assumption does not match the current state of practice observed in real software systems. Besides, none of the current techniques actually aid developers when programming their own handlers.

In this context, this paper presents the proposal and evaluation of a set of three heuristic strategies for recommending exception handling code. The goal of the proposed heuristics is to accurately find code fragments
implementing exception handling and recommend a list of these fragments ranked by relevance according to the context of developers’ implementation task. Hence, the proposed heuristics may assist developers in the process of discovering exception handling actions relevant in their development context. The recommended fragments might be used as examples of how to implement handling actions in the context of the method being implemented by the developer.

In order to perform their recommendations, the heuristics assume that by searching code fragments that are structurally similar to developers’ code, they can discover handling actions relevant to developers’ context. In other words, it is assumed that by recommending a code fragment structurally similar to developers’ code, the handling actions implemented by the fragment are more likely to be relevant in the context of developers’ code. Thus, a set of structural information is used to represent the context of developers’ code. This set contains: (i) the types of raised exceptions, (ii) the methods called and (iii) the types of variables used. These structural information, also called structural facts in this paper, are extracted from developers’ code and used by the heuristics to search for relevant code fragments. It is worth mentioning that it is not the goal of the heuristics to provide code fragments to be reused as-is. Optimistically, we expect that by exposing developers to alternative ways of handling exceptions they become more aware of exception handling policies and implement their handling actions more properly.

The remainder of the paper is structured as follows. Section II presents a motivating example. Section III defines the proposed heuristics and Section IV depicts the heuristics implementation. Section V presents the evaluation process performed. Section VI presents related work and Section VII concludes the paper.

II. Motivating Example

Consider, for instance, that the developer is working on the code depicted in Figure 1. The method `readFileContent` reads and returns as a string the content of a given file.

```java
1. public String readFileContent(File file) {
2.     FileReader reader = new FileReader(file);
3.     try {
4.         String line = reader.readLine();
5.         // process file content
6.     } finally {
7.         reader.close();
8.     }
9.     return content;
10. }
```

The method `readFileContent` depicted in Figure 1 has three unhandled exceptions. The instantiation of `FileReader` (line 2) may throw an exception of type `FileNotFoundException`, whereas the method calls `ready` (line 3) and `readLine` (line 4) may throw exceptions of type `IOException`. If the developer decides to handle the unhandled exceptions within `readFileContent`, he could take advantage of examples of common handling actions used to handle these exception types in order to decide which handling actions he might implement. Consider the code fragment in Figure 2 as an example provided to the developer implementing the method depicted in Figure 1.

```java
1. public String getResolveMessage(File file) {
2.     ...
3.     rdr = new FileReader(file);
4.     try {
5.         while ((s = rdr.readLine()) != null) {
6.             // build resolve message
7.             return message;
8.         } catch (IOException e) {
9.             StatusManager.getManager().handleStatus(STATUS.ERROR, e);
10.         } finally {
11.             rdr.close();
12.         }
13.     } catch (FileNotFoundException e) {
14.         throw new IllegalArgumentException(msg);
15.     } finally {
16.         return errorMessage;
17.     }
18. }
```

In the code example depicted in Figure 2, the information regarding exception handling that are actually useful to the developer are highlighted in dark-gray. In the `finally` block (lines 15-21), for instance, there is a call to the method `close` (line 17). This information is important to remember the developer to release previously allocated resources, avoiding typical performance issues related to resource leak [5].

In the catch blocks that handle exceptions of type `IOException` (lines 13-15 and lines 18-20), exceptions of type `IllegalStateException` are thrown. Rethrowing caught exceptions with another type is a typical handler. This type of handler is especially common when the system’s architecture is organized in layers. When an exception crosses the boundaries of two different layers, it may leak inner information about the signaler layer. In layered systems, developers are encouraged to remap crossing-boundaries exceptions in order to preserve the encapsulation of the signaler layer, avoiding typical architectural violations [6].

At last, the catch block that handle exceptions of type `FileNotFoundException` (lines 22-33) shows an error message dialog in the Eclipse graphical user interface. Although showing an error message dialog in the user interface is a fairly simple task in most software applications, it is worth noticing how this task can become quite complex when working with framework-based development. Developers implementing an Eclipse-based application must first instantiate an object whose constructor receives five parameters. Then they have to invoke two chained methods, passing to the second method two parameters, one of which is a public integer constant in other class, in order to finally display...
an error message dialog. Without adequate support, software developers implementing framework-based applications would have a harsh time in finding this sort of information [13].

III. DEFINITION OF THE HEURISTIC STRATEGIES

The goal of the proposed heuristic strategies is to assist developers in discovering handling actions that are relevant in their method’s context. The heuristics search for code fragments and rank these fragments by relevance, according to the context of developers’ method. The list of recommendations is ranked in order to place relevant examples as close as possible to the first position of the list. Thus, developers will not waste their time examining and discarding irrelevant examples.

The design of our heuristics contemplated a set of structural facts extracted from source code because they are the easiest information related to exception handling to obtain. In fact, the source code is, in most cases, the only source of information about exception handling available. Exception handling policies implemented in software systems are poorly and rarely documented in specifications, code comments, or any other software artifact. Moreover, by extracting structural facts directly from the source code it was not necessary to make any other assumptions about documentation quality or about adherence of source code to documented policies. At last, the structural facts extracted can be obtained with minimum additional overhead and without any sort of extra information.

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.

The Heuristic of Exception Type is defined as follows. Given a method $M_i$ that wants to handle an exception of type $T_j$. The Heuristic of Exception Type searches for code fragments based on their handled exception type. Assume, for instance, a fragment $M_1$ that handles an exception of type $T_2$. If $T_2$ has the same type of $T_1$, then fragment $M_2$ is selected. Also, if $T_2$ is a super-type of $T_1$, but not the root of the exceptions hierarchy tree, then fragment $M_2$ is selected. The fragment $M_2$ will only be discarded if $T_2$ is not an ascendant of $T_1$, or if $T_2$ is the root of the exceptions hierarchy tree. To rank the selected candidates, the Heuristic of Exception Type considers the distance in the hierarchy tree between the types $T_1$ and $T_2$. The closer $T_2$ is from $T_1$ within the exception hierarchy tree, the more similar $M_2$ is considered when compared to $M_1$. Figure 3 depicts an example of the Heuristic of Exception Type.

Consider the method $M_1$ in Figure 3 as the method being implemented and in which a developer wants to handle an exception of type $T_2$. The Heuristic of Exception Type will examine three code fragments implementing exception handling: $Ex_1$, $Ex_2$ and $Ex_3$. Consider also that the exception type $T_2$ is a direct super-type of $T_1$ and that the exception type $T_3$ is not an ascendant of $T_1$. In the scenario depicted in Figure 3, the fragments $Ex_1$ and $Ex_2$ are selected, whereas fragment $Ex_3$ is discarded. Fragment $Ex_1$ is selected because it handles an exception with the same type of the exception that the developer wants to handle, $T_1$. Fragment $Ex_2$ is selected because it handles an exception that is a super-type of $T_1$. And fragment $Ex_3$ is discarded because it handles an exception type that is not a super-type of $T_1$. The type of the exception handled by $Ex_1$ is an exact match of the exception type for which a handler is wanted, $T_1$, whereas the type of the exception handled by $Ex_2$ is a direct super-type of $T_1$. Thus, $Ex_1$ is considered more similar to $M_1$ than $Ex_2$. Consequently, the fragment $Ex_1$ is recommended in a better position than fragment $Ex_2$ within the recommendations list.

B. Heuristic of Methods Called

Properly handling an exception does not rely solely on its type, but also on the context of the task implemented by the method where the exception is being handled. In order to represent the task implemented by a given code fragment, the Heuristic of Methods Called assumes that code fragments that call the same methods implement similar tasks.

The Heuristic of Methods Called is defined as follows. Given a method $M_i$ that calls methods $C_1, C_2, ..., C_n$. Assume that one of the calls $C_i$, where $i=1...n$, throws an exception of type $T_i$ and the developer wants to handle this exception. The Heuristic of Methods Called searches for code fragments based on the methods called by $M_i$. Assume $M_j$, a code fragment that implements exception handling. If $M_j$ invokes method $C_i$, then it is selected. If $M_j$ does not invoke method $C_i$, it is discarded. To rank the selected fragments, the Heuristic of Methods Called compares the set of methods called by the developer’s method, $M_i$, and a code fragment, $M_j$. The more common calls $M_i$ and $M_j$ share, the more similar $M_j$ is considered when compared to $M_i$. Fragments that share the greatest number of calls with the developer’s code are placed in the first positions of the recommendations list. Figure 4 depicts an example of the Heuristic of Methods Called.

Consider the method $M_1$ in Figure 4 as the method under development in which a developer wants to handle an exception of type $T_1$. The exception of type $T_1$ is thrown by the invocation of method $c_1$. Consider also three code fragments: $Ex_1$, $Ex_2$ and $Ex_3$. In the example depicted in

![Figure 3. Heuristic of Exception Type](image)

![Figure 4. Heuristic of Methods Called](image)
Figure 4, the Heuristic of Methods Called selects the code fragments Ex₂ and Ex₃, since they both invoke the method c₁. The fragment Ex₁ is discarded because it does not invoke the method c₁. The fragment Ex₂ invokes three methods (c₀, c₁ and c₄) that are also invoked by M₁, whereas the fragment Ex₂ only invoke two methods (c₀ and c₄) also invoked by M₁. In this manner, the Heuristic of Methods Called considers the fragment Ex₃ more similar to M₁ than Ex₂. Consequently, Ex₃ will be recommended in a better position within the recommendations list than Ex₂.

C. Heuristic of Variable Types

As an alternative way of representing the context of a task implemented by a method, the Heuristic of Variable Types assumes that methods using variables of same types implement more similar tasks. The Heuristic of Variable Types is defined as follows. Given a code fragment M₁, consider V₁ the set of variables used by M₁ and TV₁ the set of types of the variables in V₁. The Heuristic of Variable Types searches for code fragments M₂ that use at least one variable whose type is in TV₁. The more common variable types M₁ and M₂ share, the more similar M₂ is considered. The operation of the Heuristic of Variable Types is similar to the operation of the Heuristic of Methods Called. For the sake of space we omit its example.

IV. HEURISTICS IMPLEMENTATION

As stated in Section III, the heuristic strategies perform their searches based on structural facts. Besides searching for candidates to recommendation, the heuristic strategies must also sort the set of selected fragments in terms of their similarity to the developers’ code. In this manner, the implementation of the proposed heuristic search strategies requires: (i) a query mechanism able to find candidates based on a set of structural facts; (ii) a ranking mechanism able to grade a code fragment according to its similarity to the developers’ code. The rest of this section is structured as follows. Section IV.A depicts the query mechanism used to implement the heuristics strategies and Section IV.B shows the ranking mechanism.

A. Query mechanism

The first responsibility of the heuristics strategies is to select code fragments based on a set of structural facts extracted from developer’s code. To implement the first responsibility of the proposed heuristics, it is necessary to build queries based on the set of structural facts. The query mechanism is responsible for building these queries.

The queries built by the query mechanism are represented as disjunctive Boolean expressions. Other ways of representing the single query are also possible by building Boolean expression that combine other Boolean operators, such as ‘AND’ and ‘NOT’. We opted for the disjunctive form because it is the less restrictive form possible, i.e., is the form that selects the highest number of candidates per query. Since all heuristics are represented as a single query, the disjunctive form allows selecting a candidate even if it does not satisfy the criteria of all heuristics. The high number of candidates selected by a query in the disjunctive form is handled by ranking these candidates according to a grading function defined by the ranking mechanism, as described in the next section.

B. Ranking mechanism

The number of candidates that satisfy one of the queries built by the query mechanism can easily overcome the limit of hundreds. Recommending all these candidates selected by the query mechanism directly to the developer would be ineffective. The developer would be overwhelmed with so much information that he would probably give up on trying to find the information that he seeks. For that reason, it is crucial to recommend to the developer only a subset of the candidates selected by the query mechanism. Preferably, a subset of candidates containing information that is more likely to be relevant to the developer in his exception handling task. Thus, it is important to identify which of the selected candidates are more likely to have relevant information.

The query mechanism initially extracts the structural facts required by the heuristics from a method for which recommendations are provided. The structural facts are extracted and represented as terms, i.e., pairs \( \langle \text{fact}, \text{value} \rangle \). For the Heuristic of Exception Type, for instance, a set of terms in the form \( \langle \text{HANDLES}, \text{value} \rangle \) is extracted. For the Heuristic of Methods Called and for the Heuristic of Variable Types the structural facts are extracted as terms with the first element named CALLS and USES, respectively.

After extracting the structural facts required by the heuristics, the query mechanism builds a query based on these facts. Although the heuristics strategies have been presented separately in Section III, they are all implemented by the query mechanism together as a single query. In this manner, it is possible to retrieve all fragments selected by each heuristic at once. The single query is built as follows.

Assume \( F \) as the set of structural facts extracted from a method. The query \( Q \) based in \( F \) is defined as the disjunctive expression of the terms in \( F \). Consider, for instance, the following set of structural facts extracted from a method:

\[
\begin{align*}
F &= \{ \text{HANDLES:IOException}, \text{USES:FileWriter}, \\
&\qquad \text{CALLS:readFile, closeFile, USES:File,} \\
&\qquad \text{USES:FileWriter, USES:BufferedWriter} \}
\end{align*}
\]

The single query \( Q \) built based on \( F \) will have the form:

\[
Q = \{ \text{HANDLES:IOException OR calls:openFile OR calls:closeFile OR uses:File OR uses:FileWriter OR uses:BufferedReader} \}
\]

The queries built by the query mechanism are represented as disjunctive Boolean expressions. Other ways of representing the single query are also possible by building Boolean expression that combine other Boolean operators, such as ‘AND’ and ‘NOT’. We opted for the disjunctive form because it is the less restrictive form possible, i.e., is the form that selects the highest number of candidates per query. Since all heuristics are represented as a single query, the disjunctive form allows selecting a candidate even if it does not satisfy the criteria of all heuristics. The high number of candidates selected by a query in the disjunctive form is handled by ranking these candidates according to a grading function defined by the ranking mechanism, as described in the next section.
The heuristics assume that candidates that are structurally similar to the developer’s code are more likely to have relevant information. In order to rank the selected candidates regarding their similarity to the developer’s code, the ranking mechanism defines a ranking function. The ranking function attributes to a code fragment \( C \) and a query \( Q \), built by the query mechanism based on a method \( C_{\text{Base}} \), a grade \( G \) that represents as a real value the similarity degree between \( C \) and \( Q \). Since \( Q \) is built based on \( C_{\text{Base}} \) (the developer’s code fragment) the ranking function indirectly computes the similarity degree between \( C \) and \( C_{\text{Base}} \). The ranking mechanism orders the candidates selected by the query mechanism based on the grades computed by the ranking function: candidates with highest grades are in top positions within the list of recommended candidates.

The ranking function is based in two factors. The goal of the first factor of the ranking function is to identify candidates who are more similar to a given query. The goal of the second factor is to allow developers to adjust the results of the heuristics to better suit their needs. They are able to do so by adjusting weights in the ranking function that are related to each heuristic strategy. In this manner, the ranking function combines two factors of distinct nature. The first factor captures the structural similarity between the developer’s code and the candidates stored in the examples repository, whereas the second factor captures the domain knowledge of the user that might refine the results provided by the heuristics.

The first factor of the ranking function is based in the quantity of terms of the query that are satisfied by a candidate. A candidate \( C \) satisfies a term \( T = (\langle F \rangle, \langle V \rangle) \) when it has a structural fact named \( F \) and valued \( V \). The first factor of the ranking function defines that candidates that satisfy more terms have higher grades. Thus, when a candidate \( C_1 \) satisfies more terms in a query \( Q \) than a candidate \( C_2 \), it means that \( C_1 \) is more structurally similar to the code fragment that originated \( Q \) than \( C_2 \).

The first factor is defined as the ratio of: (numerator) the quantity of terms of the query \( Q \) satisfied by the candidate \( C \) and (denominator) the total quantity of terms of the query \( Q \).

The second factor of the ranking function attributes adjustable weights to the fields of the terms of the queries. For each field defined by the heuristics (USES, HANDLES and CALLS) a different weight may be defined. The second factor is defined as the sum of the weights of the fields of the query \( Q \) that are satisfied by the candidate \( C \).

Finally, the ranking function is defined as the product of the first factor and the second factor. Consider, for instance, these two sets of structural facts \( \text{FACTS}_1 \) and \( \text{FACTS}_2 \), respectively extracted from candidates \( C_1 \) and \( C_2 \):

\[
\text{FACTS}_1 = \{ \text{HANDLES:IOException, CALLS:openFile, CALLS:readData, CALLS:closeFile, USES:File, USES:BufferedReader} \} \\
\text{FACTS}_2 = \{ \text{handles:SQLException, USES:Resource, USES:Connection} \}
\]

Consider the query \( Q \):

\[
Q = \text{HANDLES:IOException OR CALLS:getPath OR USES:Resource OR USES:Connection}
\]

Consider also the following setting of weights:

\[
\begin{align*}
W(\text{HANDLES}) &= 5.0 \\
W(\text{USES}) &= 1.0 \\
W(\text{CALLS}) &= 1.0
\end{align*}
\]

The ranking function is computed as follows. Initially, the first factor is computed:

\[
\begin{align*}
F_1(C_1, Q) &= \frac{1}{3} = 0.33 \\
F_1(C_2, Q) &= \frac{2}{3} = 0.67
\end{align*}
\]

Then, the second factor is computed:

\[
\begin{align*}
F_2(C_1, Q) &= \sum_{T \in \{ \text{HANDLES:IOException} \}} \frac{W(T.\text{field})}{|\text{HANDLES:IOException}|} = 1.65 \\
F_2(C_2, Q) &= \sum_{T \in \{ \text{USES:Resource, USES:Connection} \}} \frac{W(T.\text{field})}{|\text{USES:Resource, USES:Connection}|} = 2.0
\end{align*}
\]

Notice that the sum is performed for each satisfied term. Hence, \( W(\text{USES}) \) was computed twice for candidate \( C_2 \). Finally, the ranking function is computed:

\[
\begin{align*}
RF(C_1, Q) &= F_1(C_1, Q) \times F_2(C_1, Q) = 0.33 \times 1.65 = 0.54
\end{align*}
\]

In this example, in which a higher weight was assigned to the \( \text{HANDLES} \) field, candidate \( C_1 \) had a higher grade than candidate \( C_2 \). If another setting of weights is used, a different result may occur.

V. Evaluation

For the evaluation process, it was necessary a repository populated with code examples implementing exception handling. Unfortunately, available repositories of code [15, 16] did not have enough methods implementing exception handling so that the heuristics could perform their recommendations. Moreover, these repositories do not index the structural facts required by the proposed heuristic strategies. In this manner, we opted to build our own repository of examples in order to evaluate the proposed heuristics.

We opted to build our own examples repository based on Java applications hosted by the Eclipse Foundation Open Source Community. This decision was motivated primarily by the renowned quality and maturity of the projects hosted by the Eclipse Foundation. These projects also provide reasonable complete documentation and have active forums of developers, which was very helpful in the further evaluation of the heuristics implementation. We also have previous experience in developing Eclipse-based applications. In this manner, it was easier to us to judge the quality of the exception handling implemented by each application.

The source codes of the applications hosted on public repositories of the Eclipse Foundation were manually downloaded. Next, for each application downloaded, a set of
20 methods implementing exception handling randomly picked was examined. If one set of methods had more than 10 methods implementing ineffective handlers, such as empty catch blocks, handler that only logs an error message, or only prints the exception stack trace, amongst others [17], its source application was immediately discarded. The applications not discarded were further examined and a set of 11 applications was selected as source of code examples. A parser was implemented in order to extract from the selected applications code examples implementing exception handling. The parser detects and discards methods implementing trivial ineffective handlers [17]. The extraction process extracted a total of 7919 code examples, where each example is at the level of a method declaration.

The preliminary evaluation performed on the implementation of the heuristic strategies was exploratory-quantitative. The goal of this evaluation was to investigate the questions:

Q1: Are the structural facts used by the heuristics sufficient to identify structural similarities among code fragments implementing exception handling?

Q2: Does the configuration of the adjustable weights associated with the heuristics allow a more precise identification of code fragments implementing exception handling that are structurally similar?

In order to collect data that would help us to answer these questions, analytical experiments were performed. The analytical experiment performed was adapted from [19]. The experiment was designed as follows:

```
BEGIN Base_Experiment
    FOR EACH candidate in the repository DO
        Extract the structural facts from the candidate
        Use the heuristics to perform a recommendation
        Register in which position the original candidate was returned within the recommended list of candidates
    END FOR
    Return the record of positions where the original candidates were found
END Base_Experiment
```

The Base_Experiment depicted above extracts from each candidate stored in the example repository its structural facts. These structural facts are then passed to the heuristics in order to perform a query in the example repository. Next, the candidates returned by the heuristics are verified and the position in which the original candidate was returned within the recommended list of candidates is registered. It is worth noticing that in each iteration the current selected candidate is not removed from the repository. The goal of Base_Experiment is to actually check if the original candidate is found and in which position of the recommendation list it is recommended.

In this preliminary evaluation, we decided to check for an exact match between the original candidate and each recommended candidate, instead of checking the relevance of the information of each recommended candidate. This decision was made in order to keep the experiment humanly treatable and easily reproducible, since it would be necessary to perform this experiment several times during the evaluation process. We are aware that assessing an exact match between the original candidate and one of the recommended candidates does not simulate the real scenario of use of the heuristics. Nevertheless, by assessing an exact match between the original candidate and each recommended candidate we had a kind of a worst-case analysis. In this sense, it is verified if the heuristics are able to recommend the original candidate within the first ten recommended candidates. If in most cases the heuristics fail to recommend the original candidate within the first ten recommendations, it is concluded that the heuristics are not able to identify structural similarities between code fragments implementing exception handling. Even though this base experiment is careless in terms of relevance of the recommended information, its great advantage is the rapid feedback provision. This rapid feedback provision was essential to systematically perform the experiment during the evaluation process of the proposed heuristic strategies.

We set up two different scenarios for the Base Experiment in order to investigate questions Q1 and Q2. The first scenario regards to question Q1 and is further discussed in Section V.A, whereas the second scenario regards to question Q2 and is further discussed in Section V.B.

We have also assessed the content of the information provided in the recommendations. The main goal of the heuristic strategies is to provide examples of code implementing exception handling. Therefore, it is necessary to evaluate the relevance of the code fragments recommended by the heuristics. The evaluation of the recommendations’ relevance was qualitative-exploratory. The goal of this evaluation was to investigate the following questions:

Q3: Are the heuristics able to recommend relevant information for the implementation of exception handling?

Question Q3 led to a qualitative evaluation in order to verify if the heuristics actually recommend code fragments with relevant information for exception handling implementation tasks. To further investigate this question, a different experiment was set up. This third experiment is presented and further discussed in Section IV.C. At last, Section IV.D presents some lessons learned during the evaluation process.

A. First experiment – Structural facts sufficiency

The initial concern during the development process was to verify if the structural facts used by the heuristics are sufficient to identify structural similarities among code fragments implementing exception handling. This verification is crucial because the heuristics select the recommended candidates based on structural similarities. In order to investigate the sufficiency of the structural facts used by the heuristics, the Base Experiment was performed. Based on the record of the position of each candidate stored in the examples repository the histogram in Figure 5 was built.

The histogram in Figure 5 shows that the distribution of the original candidate position concentrated in the first positions. From a total of 7919 candidates, 7564 (95% of total) were recommended in the first position. For only 49 candidates (0.60% of total) they were not recommended within the first ten recommendations.
In a first glance, the result presented in the histogram of Figure 5 seems very exciting, but it is not a fair result. When the structural facts are extracted from a candidate and passed to the heuristics, the heuristics consider all facts, including the private ones. We call private facts those related to properties that are not visible outside their enclosing project. Since private facts are only visible within their enclosing project, using a private fact to search for candidates highlights only a small subset of candidates within the examples repository. In this manner, by using all facts extracted from a candidate stored in the examples repository to perform a query in this same repository, the private facts interfere positively in the heuristics’ accuracy. In a real scenario of use of the heuristics, the code fragment that the developer is implementing would not have access to private facts. For that reason, a different setting for the Base Experiment was set up in order to mitigate the noise caused by private facts. Thus, we tried to get a better indicator of the sufficiency of the structural facts used by the heuristics.

In the second setting for the Base Experiment, the extraction of the structural facts from the candidates was modified in order to exclude their private facts from the set of structural facts used by the heuristics to perform their search. Any structural fact related to properties of the same project of the candidate was considered a private fact. In the case of a candidate br.pucr.io.inf.Example.foo(), for instance, any property whose fully qualified name has the prefix br.pucr.io.inf would be considered a private fact. Figure 6 shows the histogram built with the record of the position of each candidate stored in the examples repository when the heuristics did not consider private facts.

The histogram in Figure 6 shows that the distribution of the original candidate position still concentrated in the first position. A tally of 6710 candidates (85% of total) were recommended within the first ten positions and 5535 candidates (70% of total) were already recommended in the first position. It is noteworthy that the number of candidates that were not recommended within the first ten positions of the list of recommended candidates increased considerably when compared to the result of the first setting of Base Experiment. The main reason for this increase was the own setting of the experiment. By removing all private facts from the set of structural facts used by the heuristics there were cases where very few facts remained. With small sets of structural facts the heuristics were not able to detect the original candidate. There were even extreme cases where remained no structural facts to be used by the heuristics.

These cases are represented in the histogram of Figure 6 by the bar labeled FACTS = { }. Besides, there were also cases that after removing all private facts remained only exceedingly ordinary facts, i.e., facts that are common to many candidates in the examples repository. The structural fact USES: String, for instance, is owned by almost 60% of the examples stored in the repository. When the heuristics base their searches only in exceedingly ordinary facts they are not able to differentiate the original candidate from the others, since they all have the same structural facts. 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. Thus, the heuristics were not able to recommend the original candidate within the first ten recommendations.

### B. Second experiment – Influence of weights on precision

As stated in Section IV.B, the ranking function has a factor that attributes weights that can be adjusted by the developer. In the experiment described in the previous section, the weights were set up to their default values. The default setting attributes the value 1 for each field. In this section, we describe the experiment performed in order to assess if the weights adjustment allows a more precise identification of code fragments that are structurally similar. This experiment does not aim to determine a specific “optimal” weight setting. Instead, it aims to investigate why some weight settings have better results than others. The experiment was set up as follows.

Initially, a set of weight settings was defined. To do so, a base set of weights was defined: BASE = \{1,2,3,4,5,6,7,8,9,10\}. From this base set all permutations with three elements (considering elements repetition) were generated: \{ \{1,1,1\}, \{1,1,2\}, \ldots, \{10,10,10\} \}. A total of one thousand permutations were generated. For each permutation, the Base Experiment was performed, considering the first element of the permutation as the weight of the field HANDLES, the second element the weight of the field CALLS and the third element the weight of the field USES. The private facts were removed in this experiment to avoid their noise in the results. At last, two metrics were defined in order to compare the results of each weight setting. The first metric counts the number of original candidates that are recommended in the first position of the recommendations list. This is a metric for the best-case scenario. The second metric counts the
number of original candidates that are not recommended within the first ten positions of the recommendations list. This is a metric for the worst-case scenario.

For the first metric, the weight setting \{10,3,10\} had the best result. It allowed the heuristics to recommend the original candidate in the first position of the recommendations list in 5725 of the 7904 cases (72% of total). The weight setting \{1,10,1\} had the worst result, recommending the original candidate in the first position in 5441 cases (68%). For the second metric, the weight settings \{10,5,10\} and \{9,5,10\} tied as the best setting. They were not able to recommend the original candidate within the first ten positions of the recommendations list in 955 cases (12% of total). The worst setting was the default setting \{1,1,1\}, which failed to recommend the original candidate within the first ten positions of the recommendations list in 1194 cases (15% of total).

The default setting was amongst the worst settings in both metrics used to compare the weight settings in this experiment. For the first metric, the default setting was only the 983rd best setting. For the second metric, it was the 1000th best setting, i.e., it was the worst amongst all settings. For this experiment and the examples repository used, the adjustment of weights allowed the heuristics to identify more precisely code fragments that are structurally similar. By analyzing the best and worst weight setting for both metrics we also observed a pattern in the relation of the individual weights.

For both metrics, all the best ten weight settings presented the following pattern: CALLS < HANDLES, CALLS < USES and HANDLES = USES. An opposite pattern occurred for the worst ten weight settings: CALLS > HANDLES, CALLS > USES and HANDLES ≈ USES. This curious symmetry between the best and worst weight setting caused us to thoroughly investigate this phenomenon.

We observed that many candidates stored in the examples repository invoke exceedingly ordinary methods, such as StringBuffer.toString, String.equals, Throwable.getMessage, amongst others. When a high value was assigned to the weight of the field CALLS, the heuristics highlighted similarities between the original candidate and many others candidates. Thus, the heuristics were not able to distinguish the original candidates from the others.

This phenomenon also occurred with the structural facts related to the type of the variables, since there are also exceedingly ordinary types of variables in the repository, such as String, Object, Double, amongst others. However, the type of the handled exception overshadowed it. Since the type of the exception is a parameter passed to the catch block, it was also considered as a structural fact related to the type of the variable used. In this manner, when a high value was assigned to the weight associated with the field USES, it was indirectly contributing to the Heuristic of the Exception Type, since the exception type is also considered a variable type. We observed that for the best weight settings, the weights associated with the fields USES and HANDLES were always higher than the weight associated with the field CALLS. In practice, the best weight settings had the best results due to the high weight associated with the field HANDLES. The high value assigned to the field USES was only a side effect of the fact that we also considered the exception type as a structural fact of a variable type.

C. Third experiment – Recommended information relevance

For this experiment, it was selected two Eclipse-based applications that were not used to populate the examples repository. Each application had its source code examined in order to select methods implementing exception handling to be used as input to the heuristics. The methods were manually inspected to discard any clone-methods, or methods that were very similar. From each application, 10 methods were selected. These selected methods had their exception handling code removed. Next, for each method selected, the heuristics were used to recommend a list of ten code fragments. Each code fragment was classified as ‘Relevant’ or ‘Irrelevant’.

In order to determine if the code fragment recommended was actually ‘Relevant’ or ‘Irrelevant’ we defined an oracle. To do so, a set of exception handling guidelines for Eclipse based applications was compiled. These guidelines were compiled from various documents provided by the Eclipse Foundation Open Source Community, such as tutorials, APIs, FAQs, forums and mostly from the source code of Eclipse based applications. The guidelines are described in terms of classes and methods that must be used to perform some tasks related to exception handling in the context of Eclipse-based applications. The example below depicts one of the guidelines compiled for this experiment.

Guideline 91: Release all listeners registered in an observable object, and all listeners this object might have registered on other observable objects.

Call the following code:

```java
<instance of IObserver>.dispose();
```

So, for each recommended candidate, it was considered as ‘Relevant’ if its handler implemented at least one of the guidelines. And it was considered as ‘Irrelevant’ if its handler implemented none of the guidelines. All 20 methods used as input in this experiment had at least three recommended candidates considered ‘Relevant’. Of the 20 base-methods, 12 had the first recommended candidate considered ‘Relevant’, and only 1 method had no ‘Relevant’ candidate within the first six recommendations. The only method that had no ‘Relevant’ candidate within the first six recommendations was further investigated, since it was the worst one. When the recommendations received by this method were carefully investigated, we noticed an interesting pattern. The first five recommended candidates invoked the method IResourceTree.failed( IStatus ) in their handler as follows:

```java
1. try( ... )
2. catch (CoreException e )
3. { ...
4. resourceTree.failed ( e.getStatus() );
5. }
```

When the API of the method IResourceTree.failed( IStatus ) was checked, it was confirmed that it is an specific method to handle exceptions related to operations performed by objects of type
IResourceTree. Since this information was not compiled in the oracle in the beginning of the experiment, we considered these recommended candidates as ‘Irrelevant’, although they actually had relevant information. This scenario highlights the heuristics potential to aid developers in discovering new ways of implementing their exceptions. The information provided by the heuristics strategies was the first step in the process of discovering a new handling action.

D. Lessons learned and possible improvements

The results of the experiments performed are only valid for the methods used as input in the third experiment and for the repository of examples built. We are aware that the extraction process of examples helped to improve the relevance of the recommendations, since the parser automatically discarded trivial ineffective handlers. In this manner, we avoided the recommendation of ineffective handlers. We do not consider this as a faked scenario of use, since we envision the heuristics being used with a repository of good examples. We believe that the better the information available, the better the recommendations will be, which is our primary goal. For this purpose, it might be the case that one of the developers of a development team is responsible for constantly refining and updating the repository of examples.

The oracle used to categorize the recommendations in Relevant or Irrelevant also interfered in our results. A more rigorous evaluation with software developers using the heuristics in programming tasks for considerable periods of time is needed to actually assess their usefulness. Since we still do not have a proper graphical user interface for the heuristics prototype, we could not perform controlled experiments with developers. In this context, the compilation of the oracle was an attempt to mitigate a more biased assessment based solely in our own opinion.

Despite the promising results of our experiments, during the process of defining, implementing and assessing the heuristics a few lessons were learned and some future improvements were also identified, which elaborate on in the following.

First, even if the parser used during the fact extraction process tries to detect and discard ineffective handlers, there were still some of these handlers stored in the repository used in the experiments. Most of them were a variation or a combination of typical ineffective handlers. For instance, a handler that prints the exception stack trace and also invokes the return command was not detected as an ineffective handler. With minor adjustments in the parser these ineffective handlers will also be detected and discarded during the extraction process.

Second, during the first experiment we observed that ubiquitous facts have low potential in highlighting structural similarities. Even worse, we observed during the second experiment that when high values are assigned to the weights of heuristics based on ubiquitous facts, the precision of the heuristics decays. To handle the ubiquitous facts phenomenon, we could compute the frequency of each structural fact and takes it into account when computing the ranking function, as do information retrieval techniques for text-based documents, such as tf-idf [15]. In this manner, ubiquitous facts would have a lower impact on the final grade, whereas rare facts would have a higher impact.

Finally, during the third experiment, we observed that in some cases the code examples recommended were very long methods, containing approximately 300 lines of code. Such very long methods have a big set of structural facts and may become a convergence point of recommendations. The heuristics may identify similarities with these long methods very often, since they have many different structural facts and, therefore, will have a high chance of having some structural similarities with the base-method. Thus, it is possible that very long methods be recommended very often. This convergence point of recommendations phenomenon was not actually observed during our experiments. Yet, the fact that very long methods were observed among the recommendations suggests that we should address this potential problem.

One possible improvement is to consider the size of the set of structural facts of the candidates stored in the examples-repository in the ranking function. Instead of considering the absolute number of terms of a query that are satisfied by one candidate, the ranking function could consider the ratio of the number of terms of a query that are satisfied by one candidate divided by the total number of facts owned by that candidate. A more extreme possible improvement would be to refine the fact extraction process to discard very long methods.

VI. RELATED WORK

This section presents a representative set of related work. Section VI.A presents works related to tool support for exception handling, whereas Section VI.B presents works related to approaches that aid developers in software engineering tasks.

A. Exception handling tool support

In the last decade, several techniques were proposed in order to aid developers in comprehending and maintaining exception handling elements in source code. Chang et al. [10] used static analysis techniques to estimate exceptional propagation flows in order to identify unnecessary try-catch blocks and throws statements. However, only removing unnecessary elements of the source code would not solve, or mitigate, the problems related to the poor quality of handling actions. Robillard and Murphy [11] proposed a technique to represent exceptional propagation flows. The authors claim that their technique aid developers in comprehending the global impact of the propagation of exceptions. Yet, the number of exceptional flows computed by their technique, which is in the order of thousands, is impossible to be treated by humans. Fu and Ryder [12] proposed the chained-exception static analysis technique to reduce the amount of data generated by Robillard and Murphy technique. While the technique does reduce the number of exceptional flows generated, this number is still in the order of thousands. All these work related to tools that aid developers in exception handling tasks have a different goal than our work. The other works aim at aiding developers in maintenance and comprehension tasks of a previously implemented code, whereas our work aims at aiding developers in implementing their exception handling code from the beginning.
B. Tool support for software engineering tasks

The amount of information involved in software engineering tasks has greatly increased due to the high complexity of current software systems. In recent years, several tools were proposed that aim at assisting developers in coping with this increased complexity during some software engineering tasks.

Holmes, Walker and Murphy [18] propose the Strathcona recommender system, which aids developers in learning how to use frameworks and APIs by providing concrete code examples. Strathcona extracts structural information from the developer’s code, such as method signature and variables type and identifier, in order to perform a heuristic search in a local repository of examples. This approach is very similar to ours. The main difference relies on the structural information used by each approach. In particular, our proposed heuristics consider structural information regarding the exception type as a different fact, whereas their approach does not differentiate the exception type from the types of other common variables. With this more detailed information, our approach is able to differentiate a method that handles a specific exception type, from a method that throws this same exception type, for instance. In this manner, our heuristics are able to recommend code examples implementing exception handling more precisely.

Bruch, Mezini e Monperrus [19] propose an approach for mining auxiliary framework documentation from source code of client applications. The auxiliary documentation is provided in terms of framework usage directives. This approach is similar to ours in the sense that it aids developers in discovering. However, our approach focus in providing concrete code examples, whereas their approach focus in providing guidelines of usage. Moreover, their approach does not provide any sort of usage directive regarding exception handling.

Buse and Weimer [13] propose a technique for mining and synthesizing documentation of program interfaces. This documentation is provided in the form of code snippets examples, which also include exception handling code in some cases. However, the authors were not careful in the process of extracting their examples in order to ignore poorly implemented exception handling code. Thus, their technique might recommend poorly implemented handlers very often.

VII. FINAL REMARKS

Exception handling mechanisms have been proposed since the late 1970’s. Since then, they have not changed much. Although the necessary time for software technology transfer has already passed, the state of the practice of exception handling in real software systems shows that implementing exception handling code is still a daunting task for most developers. We believe that most problems with exception handling observed are more related with developers’ lack of proper training, than with the exception handling mechanisms themselves.

In this paper we proposed and evaluated a set of heuristic strategies used to recommend concrete code examples implementing exception handling code. The goal of the proposed heuristics is to assist developers in the process of discovering relevant handling actions. Our preliminary assessment shows that the heuristics proposed have potential in providing code examples containing relevant information for exception handling implementation tasks. The heuristics proposal and evaluation is the first step towards our main goal: build a recommender system for exception handling code integrated to developers’ development environment. We are currently implementing a new version of the heuristics prototype as an Eclipse plugin. We intend to use this more complete and user-friendly version of the heuristics prototype to perform controlled experiments with real developers. The goal is to check the usefulness of the recommendations in scenarios simulating real exception handling tasks.

REFERENCES