2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

On the Use of Context in Recommending Exception Handling Code Examples

Mohammad Masudur Rahman Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
{masud.rahman, chanchal.roy}@usask.ca

***Abstract*—Studies show that software developers often either misuse exception handling features or use them inefficiently, and such a practice may lead an undergoing software project to a fragile, insecure and non-robust application system. In this paper, we propose a context-aware code recommendation approach that recommends exception handling code examples from a number of popular open source code repositories hosted at GitHub. It collects the code examples exploiting *GitHub code search API*, and then analyzes, filters and ranks them against the code under development in the IDE by leveraging not only the *structural* (i.e., graph-based) and *lexical* features but also the *heuristic quality measures* of exception handlers in the examples. Experiments with 4,400 code examples and 65 exception handling scenarios as well as comparisons with four existing approaches show that the proposed approach is highly promising.**

***Index Terms*—Exception handler; context-relevance; lexical similarity; structural similarity;**

I. Introduction

Exception handling is one of the most important tasks that software developers undertake during software development and maintenance. However, studies show that developers either use the exception-handling features ineffectively [15] or misuse them in the real life software development [26]. Cabral and Marques [15] conduct a study with 32 applications from Java and .NET frameworks, and report that about 40%-70% exception handling actions are overly simplified or ineffective. The actions either log error messages and print stack traces or simply do nothing. According to their findings, developing effective exception handlers is a daunting task. One way to benefit both the developers' productivity and the quality of the exception handlers is to recommend readily available and relevant exception handling code examples to the developers within the scope of their working environment (e.g., IDE), which can be leveraged in handling exceptions by them.

A number of existing studies on exception handling attempt to support the developers through useful insights from static analysis of the exception control flows and handling structures [16, 18, 24] or comparative field studies [15, 17], visualization [26], and recommendation of code examples [13]. Barbosa et al. [13] propose an approach to recommend exception handling code examples exploiting the structural facts of the code under development and the candidate examples. Although the approach performs considerably well on their carefully constructed dataset, it suffers from several limitations. First, the approach considers only the usage of certain API classes and API methods, and captures neither static relationships (i.e., method belongs to which class) nor the dependencies among different API objects used in the code. These static or dependency relationships can be considered as an important structural component of a code example. Second, the constructed dataset is static and cannot be easily updated. It also requires significant amount of manual preprocessing to be useful in the recommendation for exception handling.

In this paper, we propose a *context-aware* recommendation approach for exception handling code examples, which leverages not only both the *structural* and *lexical* features but also the *quality of the exception handlers* in the code examples. The approach exploits the *GitHub Code Search API* [2], and collects about 60-70 code examples from GitHub code repositories against a search query representing the context code (i.e., code under development) in the IDE and the exception a developer attempts to handle. It then analyzes, filters and ranks the examples based on their relevance against the context code and the quality of their exception handlers.

The proposed approach also overcomes certain limitations of the existing techniques. First, it adopts a *graph-based technique* for structural relevance estimation, where the approach identifies all the API objects along with their static relationships and data dependencies in the code to develop an *API usage graph*. We believe that two code fragments having similar usage graphs (i.e., similar set of API objects with similar static or dependency relationships) are likely to accomplish similar programming tasks. The usage graph captures more useful and more in-depth structural features of the code compared to existing structural heuristics [13, 19]. We thus exploit the usage graph matching idea for structural relevance estimation (i.e., novelty of our approach), and it helps to overcome the limitations of the heuristic-based techniques. Second, it applies a state-of-the-art lexical feature-based code cloning technique [25] in order to determine the *lexical similarity* between context code in the IDE and the candidate examples, which was completely ignored by the existing studies. The idea is to recommend the code examples which are not only structurally relevant but also lexically similar (i.e., easy to work with) to the context code. Third, the approach integrates one of the largest and the most popular online code bases, *GitHub*, into the IDE, which can provide readily available exception handling code examples from the top ranked repositories. This integration makes the corpus for recommendation dynamic, constantly evolving, and synchronized with a number of mature and popular projects.

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.15

```
//more code ...
try {
       URL url=new URL(WEB_SERVICE_URL_WITH_PARAMS);
       HttpURLConnection conn=(HttpURLConnection)url.
            openConnection();
       //more code goes here ...
    } catch (Exception e) { }// generic exception handler
```

Listing 1. Code under Development

We conduct experiments on the proposed approach using 4,400 GitHub code examples and 65 exception handling scenarios (i.e., each scenario consists of an exception and a code segment). The exceptions and associated context code are collected from different online sources such as StackOverflow Q & A site and Pastebin [8]. First, we perform an extensive search into GitHub code repositories using the code search feature, and develop an *oracle* by collecting the most relevant exception handling code examples for each case (i.e., scenario). We then use the oracle in order to evaluate the proposed approach, where our approach recommends relevant code examples with a maximum of 41.92% *mean average precision*, 31.07% *mean precision*, 76.70% *recall* and 86.15% *recommendation accuracy*. These results are found promising according to the existing relevant studies [20, 23, 28]. We also compare with four popular existing approaches– Barbosa et al. [13], Holmes and Murphy [19], Takuya and Masuhara [27] and Bajracharya et al. [12], for the same dataset, and find that our approach outperforms them in all corresponding performance metrics. Thus we make the following technical contributions in this paper.

- We propose a graph-based approach in order to estimate the *structural relevance* between two code segments.
- In the ranking of exception handling code examples, we not only combine *structural relevance* and *lexical relevance* but also consider the *quality of the exception handlers* in the examples.
- We package our solution into a tool, *SurfExample* [9], that captures the context code in the IDE, and recommends relevant exception handling code examples collecting from a remote web service [9], and the service can be leveraged by any IDE.

II. Motivating Example

Let us consider a problem solving scenario, where a developer implements a client module of an Eclipse plugin that accesses a remote web service. Like many other developers, she is concerned about the functional requirements, and uses only a *generic handler* for exception handling (e.g., highlighted in Listing 1). The implementation serves the primary purpose (e.g., accessing information) of the client module; however, it also poses several threats to future maintenance and evolution of the plugin. First, the generic handler catches all exceptions triggered from within the *try* block and *suppresses* each of them, which clearly violates the second accepted principle [5] of exception handling–*if you catch an exception, do not swallow it*. The suppression conceals important information of the occurred exceptions, and identification or fixation of a bug in a multilayer application with such poorly

```
BufferedReader breader=null;
try {
       URL url = new URL(this.web_service_url);
       HttpURLConnection httpconn = (HttpURLConnection) url
            .openConnection();
       httpconn.setRequestMethod("GET");
       if (httpconn.getResponseCode() == HttpURLConnection.
            HTTP_OK) {
               breader = new BufferedReader(new
                    InputStreamReader(
                    httpconn.getInputStream()));
                String line = null;
                while ((line = breader.readLine()) != null)
                    {
       //more code goes here ...
            }
         }
    } catch (MalformedURLException mue) {
      Log.warn("Invalid URL " + this.web_service_url, mue);
      MessageDialog.openError(Display.getDefault().
           getShells()[0],
                "Invalid URL " + this.web_service_url, mue.
                     getMessage());
    } catch (ProtocolException pe) {
      Log.warn("Protocol Exception " +
           this.web_service_url, pe);
      MessageDialog.openError(Display.getDefault().
           getShells()[0],
                "Invalid Protocol " + this.web_service_url,
                     pe.getMessage());
    } catch (IOException ioe) {
      Log.warn("Failed to access the data " +
           this.web_service_url, ioe);
    } finally {
      breader.close();
   }
}
```

Listing 2. Recommended Code Example

designed handlers is highly error-prone and time-consuming. Second, exceptions are generally associated with different API methods (according to API design specifications), and several exceptions can occur from a programming context. Each of those exceptions (especially checked exceptions) deserves a specific treatment (e.g., handle or rethrow) based on the application context or abstraction. The generic handler without any cleanup operations fails to meet the individual exception-specific requirements [16], and thus leads to different hidden bugs and resource or security issues.

Now let us consider the code example in Listing 2 recommended by the proposed approach for the current programming context (i.e., code under development) in Listing 1. The example performs a similar type of programming task using a similar set of API objects, and thus is completely relevant to the current context. The code example treats each of the exceptions that can trigger from the code, and it also provides valuable information for exception handling. First, the developer is often not aware of the exceptions which might occur from the current programming context. She also might not be sure of which of the exceptions are to be caught and handled if the IDE suggests them based on API specifications. The recommended relevant example provides such information, and she can easily apply that in the current context. Second, she might also lack necessary skills required to handle the exceptions, and the example demonstrates how certain exceptions should be caught and handled (e.g., highlighted lines in Listing 2). For example, the technical details of a *MalformedURLException* can be used to warn a user about the

URL, and thus it is a candidate exception for handling according to the first principle [4, 5] of exception handling–*Always catch only those exceptions that you can actually handle*. The example handles it through reporting to the user using a dialog box (i.e., instant notification) and logging the details for future maintenance. In practice, effective handling of exceptions is a frequently misunderstood aspect of programming especially with applications of multilayer abstraction, and such exception handling code examples can act as a helpful learning tool for the developer. The examples are not necessarily meant for reuse; however, they can guide her towards effective handling in her application context through exemplary implementation.

III. Background

A. Static Relationship & Data Dependency

Nguyen et al. [22] propose a graph-based approach for *API usage pattern* extraction, where they represent the usage of different API objects in the code using graphs. In the graph, each API object and its properties such as *fields*, *constructors* and *methods* are represented as nodes, and static relationships between the object and its properties or its dependencies on other objects are represented as connecting edges. They classify the dependencies into two– *data dependency* and *temporal usage order*. If an API object accepts an instance or an attribute of another object as the parameter either in the constructor or in the method, the first object is said to be dependent on the second object by data. On the other hand, certain methods can be invoked only after the invocation of another method from the same object. For example, all method invocations of an object are followed by the initialization (i.e., <*init*> method) of the object, and this type of dependency of sequence is termed as the temporal usage order. In this research, we leverage the *static relationships* between API objects and their properties as well as the *data dependencies* among different objects in the code in order to determine the structural relevance between two code segments. Fig. 1 shows the static relationships (i.e., solid edges) and the data dependencies (i.e., dashed edges) in the recommended code example in Listing 2, which uses four API classes– *URL, HttpURLConnection, InputStreamReader* and *BufferedReader*. We note that *InputStreamReader* object accesses *getInputStream* method of *HttpURLConnection* object, and *BufferedReader* object accepts an instance of *InputStreamReader* class in their constructors respectively, and these data dependencies are shown using dashed edges. On the other hand, object to property (e.g., method, constructor) static relationships are represented as green coloured solid edges.

B. Graph Matching

In graph theory, *matching graphs* involves matching a set of independent edges along with their vertices [3]. Fig. 1, the adapted API usage graph of our toy example (Listing 2), shows the static relationships and the data dependencies in the code in terms of vertices and edges. In our research, we estimate the *structural relevance* between a candidate code example and the context code, where we determine matching between

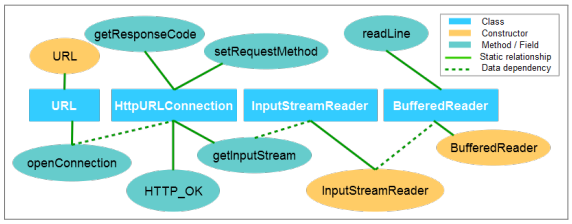

Fig. 1. Static Relationship and Data Dependency in Listing 2

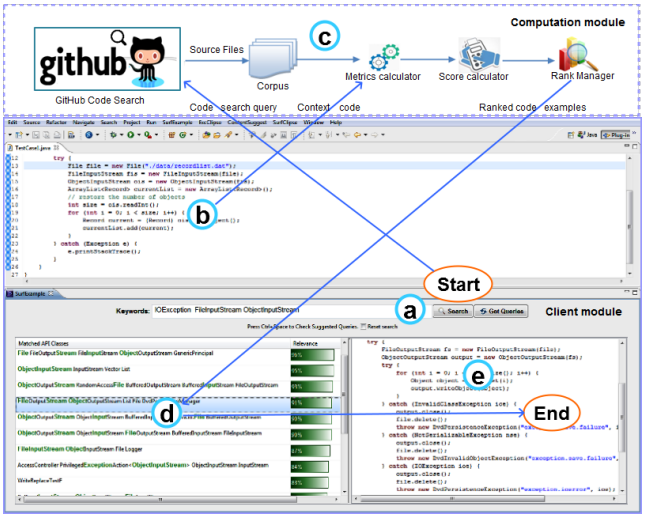

Fig. 2. Schematic Diagram of Proposed Approach

two such corresponding usage graphs. We consider *maximum matching* in the graphs, and also estimate different *heuristic weights* for different types of matching (e.g., dependency, static relations) using a machine-learning based approach (Section IV-D). For example, a data dependency matching is considered more important (i.e., contains greater weight) than a static relationship matching for relevance estimation. The static relationship matching between two graphs explains that two graphs merely contain similar set of API objects with their properties (e.g., method or field). On the other hand, the data dependency matching explains that those API objects also interact with each other in a similar fashion in both graphs, which adds more value in relevance estimation.

IV. Proposed Approach

Fig. 2 shows the schematic diagram of our proposed approach for exception handling code example recommendation. We package our solution into an easily accessible web service [9] and an Eclipse plugin [9]. In this section, we discuss different working modules of the solution, proposed metrics for relevance estimation between code segments as well as the quality estimation of the exception handlers, metric weight estimation techniques and ranking algorithms.

A. Working Modules

The proposed solution adopts a client-server architecture, and it has two working modules– *client module* (Fig. 2-(a, b, d, e)) and *computation module* (Fig. 2-(c)). The client module, an Eclipse plugin prototype [9], collects the code

under development (hereby we call it *context code*) containing generic or poorly designed exception handlers from the IDE, and prepares a search query by extracting suitable keywords from the code (Section V-B). It then sends the search query as well as the context code to the computation module. The computation module, hosted as a web service [9], collects code examples from *GitHub code repositories* using that search query and *GitHub code search API*, and develops a dynamic corpus (Fig. 2-(c)). The corpus generally contains about 60-70 code examples from hundreds of repositories, which are analyzed, filtered, and then ranked against the context code using the proposed metrics (Section IV-B) and ranking algorithms (Section IV-C). Once the ranked examples are returned from the computation module, the client module recommends the top 15 of them in the IDE (Fig. 2-(d, e)). The developer then can check the code examples and leverage for exception handling in her own programming context.

B. Proposed Metrics

This section discusses our proposed metrics which are used to determine the *structural* and *lexical relevance* of a candidate code example in the corpus with the *context code* in the IDE. It also discusses our proposed metrics that estimate the *quality of the exception handlers* in the code example.

1) Structural Relevance (R_{str}): Barbosa et al. [13] apply heuristic strategies on three structural facts–(1) the hierarchy level of the handled exception, (2) list of methods called, and (3) types of the variables used, of an exception handling code example for structural relevance estimation. Holmes and Murphy [19] also adopt a similar approach to capture the structural information from the code. They develop six heuristic strategies associated with class inheritance, method call and variable usage. Thus, existing two studies [13, 19] basically consider number of matched method calls and number of matched variables as the core components of *structural relevance* between two code segments.

In our research, we propose a *graph-based approach* (adapted from the approach of Nguyen et al. [22] for *API usage pattern extraction*) to capture the structural features from the code. We consider a code example as a network or graph of API objects interacting with each other through method or constructor invocations and field accesses in order to solve a programming problem. We consider each of the API objects, the static relationships between an API object and its fields or methods, and the data dependencies of the object upon other API objects in the code as the structural features, and we exploit them to estimate the structural relevance (i.e., structural similarity) between two code segments. Thus the structural relevance is based on four structural aspects– matched API objects, matched field accesses, matched method calls, and matched data dependencies.

API Object Match (AOM**)**: In the code, different API objects are initialized, and their fields and methods are accessed in order to accomplish a programming task. We use *JavaParser* [6], an Eclipse AST-based parser, to extract the API objects from the context code and the candidate code examples collected from GitHub. *API Object Match* metric determines the number of matched API objects between the context code and a candidate example. Given that each API object has a predefined set of fields and methods, the metric can be considered as a rough estimate of the functional similarity between the two code fragments.

Field Access Match (FAM**)**: The metric determines the matching between field accesses of an API object in the context code and that of the target object in the candidate code. While existing approaches [13, 19] ignore this feature, we use it as a structural component of the code. In practice, the metric accumulates field matching in the candidate code for all API objects in the context code, and indicates the extent to which both code fragments access the common attributes.

Method Invocation Match (MIM**)**: We consider method invocations as an important structural component of the code as the API objects generally interact with each other through them. Existing approaches [13, 19] do not consider the scope (i.e., API class instances) of the invoked methods during matching, and thus their method invocation matching might be erroneous (i.e., same method names are available in different API objects). In our research, we treat each API object as a working unit. We consider the invoked methods from an API object in the context code, and then determine the method invocation match by comparing with the invocation list from the same object in the candidate code example. In practice, the metric considers each API object in the context code and accumulates the invocation match measures.

Data Dependency Match (DDM**)**: The API objects in the code depend on each other for object initialization, method parameters and so on, and we call it *data dependency* among the objects [22]. We consider the data dependency as a structural component of the code, and we use API usage graph in order to determine the dependency matching. For example, in Fig. 1, the dependencies among the API objects are represented as dashed edges among the vertices. Given that API libraries are designed with certain dependencies among different API classes, we capture and exploit such dependencies in order to determine the structural relevance between the context code in the IDE and a candidate code example.

We sum up the above four structural components in order to determine the structural relevance score (R_{str}) as follows:

$$R_{str} = \alpha \times N + \beta \times \sum_{i=1}^{N} \frac{FAM_i}{FAQ_i} + \gamma \times \sum_{i=1}^{N} \frac{MIM_i}{MIQ_i} + \delta \times \sum_{i=1}^{M} DMT_i \quad (1)$$

Here, N and M refer to number of matched API objects and number of matched dependencies respectively. α, β, γ and δ are the weights of API Object Match (AOM), Field Access Match (FAM), Method Invocation Match (MIM), and Data Dependency Match (DDM) metrics respectively. The weight estimation technique is discussed in Section IV-D. FAQ_i and MIQ_i are number of field accesses and number of method invocations of an API object from the context code, and DMT refers to the matching weight of each data dependency. For

example, if an API object in the candidate code depends on another object through a different access point (e.g., method, constructor) than that in the *context code* (i.e., code under development), we call it partial matching (i.e., weight 0.5). On the other hand, a complete matching (i.e., weight 1.0) should match both the access points and the target end objects.

2) Lexical Relevance (R_{lex}): While *structural relevance* exploits certain API object-based structural features in the code, *lexical relevance* captures even finer level granularity– *token*. In order to capture token-level relevance between two code fragments and to add more value to relevance estimation, we use two lexical similarity measures– *cosine similarity* [1] and *code clone measure*. They also help to overcome the limitations with non-compilable code (i.e., structural relevance estimation requires the code to be compilable). Cosine similarity focuses on occurrence and frequency of a particular token in the code irrespective of its order, and thus determines the content similarity between two code segments. On the other hand, the code clone measure depends on the clone detection algorithms. In the case of cosine similarity calculation, we consider a code fragment as a vector of tokens, and discard insignificant tokens (e.g., punctuations). We then determine the cosine of the angular distance (i.e., cosine similarity) between the two such vectors corresponding to the context code and a candidate code example. In case of code clone measure (S_{ccm}), we use a state-of-the-art *code clone detection* technique, NiCAD [25], where we determine the *longest common subsequence* of tokens between the context code and the candidate code, and then normalize it as follows:

$$S_{ccm} = \frac{|S_{lcs}|}{|S_{total}|} \tag{2}$$

Here, S_{lcs} denotes the longest common subsequence of tokens, and S_{total} denotes the set of tokens extracted from the context code. The measure values between zero to one, and it provides an estimate of the extent to which the candidate code matches with the context code lexically. Thus, the two measures compute the lexical similarity between code from two different viewpoints, and we use them in order to determine the lexical relevance score (R_{lex}) as follows:

$$R_{lex} = \lambda \times S_{cos} + \sigma \times S_{ccm} \tag{3}$$

Here λ and σ are the weights of the corresponding measures, and they are calculated using a machine learning technique involving logistic regression (Section IV-D).

3) Quality of Exception Handler (Q_{ehc}): The metrics discussed earlier focus on the relevance of an exception handling code example for recommendation; however, *relevance* alone is not sufficient enough for effective recommendation (i.e., a limitation of existing studies). The quality of the exception handlers in the code is also an important concern. In addition to the above metrics and measures, we thus also consider the quality of the exception handlers in the code as follows:

Readability (RA**)**: Readability of software code refers to a human judgement of how easy the code is to understand [14]. In our research, we consider *readability* as one of most important quality metrics for an exception handler in the code example. The baseline idea is– *the more readable and understandable the handler code is, the easier it is to leverage in exception handling*. Buse and Weimer [14] propose a code readability model trained on human perception of readability and understandability. The model uses different textual features (e.g., length of identifiers, number of comments, line length) of the code that are likely to affect the human perception of readability. It then predicts a readability score on the scale from zero to one, inclusive, with one describing that the code is highly readable. We use the readily available library [10] by Buse and Weimer to calculate the readability metric of the exception handling code examples.

Average Handler Actions (AHA**)**: The metric calculates the average number of statements (i.e., actions) in each of the catch clauses in the code example. During calculation, we discard the insignificant statements such as the statements printing stack traces or error messages. We consider the measure as an important indicator of *how extensively (i.e., meaningfully) data from the caught exceptions are used for handling*. The lower the measure, the poorer the design of the exception handlers.

Handler to Code Ratio (HCR**)**: The metric refers to the fraction of the code in the example that is intended for exception handling, and we use *SLOC (Source Lines of Code)* for the calculation. While the metric indicates the *richness of the code example in handling exceptions*, it also helps to filter out the examples with poorly designed exception handlers (e.g., generic handler with empty catch block) or long methods. These examples would necessarily contain a large number of program statements compared to the handler statements in the catch clauses, and we use *Handler to Code Ratio* metric to penalize such code examples.

We use the above three quality estimates focusing on distinct aspects, and determine an overall quality estimate for the exception handlers in the code example as follows:

$$Q_{ehc} = \mu \times R + \epsilon \times AHA + \kappa \times HCR \tag{4}$$

Here, μ, ϵ and κ are the weights of the corresponding quality metrics, which are calculated using a machine learning technique involving logistic regression (Section IV-D). While HCR metric is likely to encourage examples with excessive handling code, AHA metric ensures that the handlers contain meaningful statements, and RA metric penalizes code with too many parentheses [14] (i.e., code with too many handlers).

C. Result Scores and Ranking

In our research, we consider three important aspects– *structural relevance, lexical relevance* and *quality of exception handler* for ranking and recommendation of code examples. The *structural relevance* helps to recommend a code example that uses a set of API objects similar to that of the context code in the IDE. Moreover, it ensures that each API object in that set matches with that in the context code in terms of field access, method invocation and data dependency upon other objects. The *lexical relevance* refers to the lexical similarity of a code example against the context code, and it helps to recommend similar type of code examples for possible reuse. The last aspect focuses on the overall quality of the

handlers in the code example. It helps to recommend code examples that are highly understandable, and contain good quality handlers for the exceptions of interest. Thus, the *total relevance* (R_{total}), for each candidate code example is calculated using the component scores associated with those three aspects in Equation (5). The component scores belong to different ranges due to heterogeneous feature values, and each score is *normalized* between zero to one.

$$R_{total} = w_{str} \times R_{str} + w_{lex} \times R_{lex} + w_{ehc} \times Q_{ehc} \quad (5)$$

Here, R_{str}, R_{lex} and Q_{ehc} are *structural relevance*, *lexical relevance* and *quality of exception handler* estimates respectively of a candidate code example. w_{str}, w_{lex} and w_{ehc} are the heuristic weights (i.e., relative importance) of the corresponding metrics, which are calculated using the machine learning approach discussed in Section IV-D. Once we calculate the total scores, we sort the code examples based on their scores, and recommend the top 15 examples to the developer. For instance, the code example in Listing 2 shows these values for the proposed nine individual metrics– *AOM*=2, *FAM*=0, *MIM*=1, *DDM*=0, S_{cos}=0.67, S_{ccm}=0.58, *RA*=0.09, *AHA*=1.67, *HCR*=0.52 during ranking. Given the limited (i.e., a few statements) context code (i.e., code under development) in Listing 1, the example in Listing 2 matches with it the best both structurally and lexically among all other examples. More importantly, exceptions in the example are handled carefully with at least two actions (i.e., statements) in each handler and the code is moderately readable, and thus the example gets a normalized handler quality score of 0.68. While other approaches either *analyzes manually* or *depends on the reputation* of the code repository for good quality handlers of exceptions, we not only choose reputed repositories and but also propose and use several metrics to ensure quality of the exception handlers (i.e., effectiveness shown in Fig. 3). Based on the three aspects (*structual, lexical* and *handler quality*) considered, the example scores the highest, and ranks the top in the recommended example list.

D. Metric Weight Estimation

In order to determine the weight of *nine* of the individual metrics associated with structural relevance, lexical relevance and handler quality of a code example, we choose 650 code examples handling 65 exceptions from experiment dataset. For each exception, we collect ten random candidate examples from the corpus, analyze their content, and manually tag them either as *relevant* or *irrelevant* for recommendation. We also collect the values of all nine proposed metrics for each tagged code example. We then feed the feature (i.e., metric) values and class labels (i.e., tag of example) to *Weka* tool [11] that returns a logistic regression based classifier model [7]. In the classifier model, each of the features is associated with certain coefficients, which the tool tunes in order to classify a sample (i.e., code example) with maximum accuracy. We believe that these coefficients are an estimate of the importance of the features used in the classification, and we consider them as the weights of the corresponding nine relevance and quality metrics [21]. However, the coefficients are either positive (i.e., supporting for a particular class) or negative (i.e., discouraging for a particular class), and one may find them counter-intuitive for weight estimates. Therefore, we use *Odd Ratio* of each feature, a logarithmic transformation of the coefficient, as the weight estimate for the corresponding relevance and quality metrics. Among the nine weight estimates, weights of lexical measures dominate others; that means lexical metrics play a decisive role in the classification of the code examples. Weight estimates, and associated data can be found online [9].

Once we calculate the subtotal scores using the individual metrics and their corresponding weights, they represent certain aspects such as *structural relevance, lexical relevance* and *exception handler quality* of a code example. We then adopt the same machine-learning technique (as in case of individual metrics above) in order to estimate the relative weights (i.e., importance) of those three aspects. We consider a heuristic relative weight of 1.0152 for *lexical relevance*, 1.2787 for *structural relevance*, and 1.1588 for *exception handler quality* estimate based on the *Odd Ratios* of the corresponding metrics in classifier model.

V. Experiment

A. Dataset Preparation

We collect 65 exception handling cases (i.e., scenarios) for the experiments, where each case comprises of a *context code segment* and an exception to be handled. Most of the cases are collected from different online sources such as Pastebin [8] and StackOverflow Q & A site, and a few of them are developed by us. For each of the cases, the context code is analyzed to prepare a suitable search query (Section V-B), which is then used to develop a corpus of candidate code examples containing handlers of the corresponding exception. In order to collect examples, we choose four popular software organizations–*Apache, Eclipse, Facebook* and *Twitter*, and they host about 738 open source Java projects (visited on January, 2014) at GitHub. The code bases of the target organizations are considerably rich and matured, and some of the organizations even developed exception handling frameworks (e.g., ExceptionUtils and Camel by Apache). Thus we believe that their code bases are more likely to contain code examples with efficient handlers for exceptions. We use *GitHub Code Search* and the prepared search queries to collect the code examples. For each of the cases, we collect 60-70 candidate code examples containing exception handlers, and the whole corpus contains about 4,400 examples in total.

B. Search Query Formulation

During corpus development, we prepare a search query for each of the exception handling cases, and collect the candidate code examples from GitHub code search using that query. Each of those queries generally contains two types of information–*exception name* and *dominant API class name*. We analyze the context code to extract such information, where we experience two exception handling scenarios. In the first scenario, the context code specifies which exception to be handled, and we use that exception name in the search

query. In the second scenario, the context code either does not specify the exception or contains a generic exception handler (e.g., Listing 1), and we adopt a careful approach to choose an exception (to be handled) for this scenario. Given that exceptions are associated with different API methods (according to API design specifications), we consider all the checked exceptions those might be thrown from within the context code, and choose the one that is the most frequent with the API methods in the code. In case of dominant API class name token in the search query, we analyze the API objects used in the context code. The idea is to identify the most active API objects in the code, and we consider an object with the most frequent method invocation and field access as the most active API object. Thus the search query for the context code in Listing 1 is– *IOException URL.*

C. Exception Oracle Development

We develop an *oracle* that returns a list of the most relevant code examples for each of the exception handling cases. For oracle development, we analyze code examples in the corpus collected for each case, and check for their relevance against the corresponding context code and the exception of interest. Given that checking the relevance of a code example against an exception and its context code is a subjective approach, and a number of examples are associated with each case, we use tool support in our analysis. First, we rank the examples based on their lexical similarity against the context code, and then manually check them from the top for relevance. We consult the best accepted practices [4] for exception handling, look for meaningful actions (e.g., cleanup, rethrow, status notification) other than logging in the exception handlers of a code example, and use our best judgement to choose the relevant examples. Once the examples are chosen for the oracle, they are cross-validated by the peers (e.g., two graduate research students with at least five years of Java programming experience), and we finalize the example list through discussion. We choose 176 code examples as the most relevant ones for 65 exception handling cases. It took about 50-60 working hours. The code examples are hosted online [9], and we use them as the benchmark examples to determine the performance of the proposed and existing approaches.

D. Performance Metrics

Our proposed approach profoundly aligns with the research areas of information retrieval and recommendation systems. We thus use a list of performance metrics for evaluation from those areas as follows:

Mean Precision (MP): *Precision* determines the percentage of the results (i.e., code examples) returned by a query (i.e., exception handling case) that is relevant. *Mean Precision* averages that percentage for all queries in the dataset.

Mean Average Precision at K (MAPK): *Precision at K* calculates *precision* at the occurrence of every relevant result (i.e., relevant code example) in the ranked list. *Average Precision at K (APK)* averages the *precision at K* for all relevant results in the list for a query (i.e., exception handling case).

TABLE I
EXPERIMENTAL RESULTS

Metric	Top 5	Top 10	Top 15
MP	**31.07%**	18.62%	13.85%
MAPK	**41.92%**	39.92%	38.64%
TEH[1](65)	48(101)	53(121)	**56**(135)
PEH[2]	73.85%	**81.54%**	**86.15%**
R[3]	57.39%	68.75%	**76.70%**

[1]No. of exceptions handled, [2]% of all exceptions handled, [3]% of relevant examples recommended

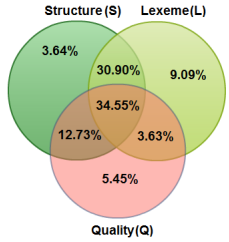

Fig. 3. Result Distribution over Metrics

Mean Average Precision is the mean of *average precision at K* for all queries in the dataset.

Recall (R): *Recall* denotes the fraction of all the relevant results (i.e., benchmark examples) that are retrieved.

E. Experimental Results

We conduct experiments with 65 exceptions (related to standard Java development) along with their context code segments, and collect the top 15 recommended code examples for each of the exceptions for evaluation. We analyze the results and determine the performance using necessary metrics (Section V-D). This section discusses the experimental results and the recommendation performance of our approach.

Table I shows the results of the experiments conducted on the proposed approach, where we apply different performance metrics such as *Mean Precision (MP), Mean Average Precision at K (MAPK), Total Exceptions Handled (TEH), Percentage of all Exceptions Handled (PEH)* and overall *Recall (R)*. We collect the top 5, top 10 and top 15 code examples from the recommendation list for evaluation. From Table I, we note that the approach provides results with 31.07% *mean precision*. That means, on average the technique recommends 31.07% relevant code examples for each of the exception handling cases, and it recommends correctly for 86.15% of the exceptions. It also successfully recommends 135 out of 176 benchmark relevant examples, which gives an over all *recall* of 76.70%. More interestingly, our approach recommends relevant code examples for 48 (73.85%) out of 65 exceptions with 41.92% *mean average precision* even when only top 5 results are considered. These results are also found promising according to relevant existing studies [20, 23, 28].

Fig. 3 shows the distribution of the *handled* (i.e., code examples correctly recommended) exceptions over different metrics–*structural relevance (S)*, *lexical relevance (L)* and *exception handler quality (Q)*. The distribution over a metric means that a certain fraction of the exceptions are handled (i.e., relevant code examples recommended) considering that metric in isolation. We note that the handled exceptions are

TABLE II
EXPERIMENTAL RESULTS ON DIFFERENT SCORE COMPONENTS

Single aspect	Metric	Top 5	Top 10	Top 15	Combined aspects	Metric	Top 5	Top 10	Top 15
Structure (R_{str})	MP	27.07%	16.76%	12.51%	Structure (R_{str}), and Content (R_{lex})	MP	27.99%	17.99%	13.44%
	MAPK	38.07%	33.84%	32.64%		MAPK	43.08%	38.69%	37.33%
	TEH(65)	45(88)	49(109)	53(122)		TEH(65)	45(91)	49(117)	53(131)
	PEH	69.23%	75.38%	81.54%		PEH	69.23%	75.38%	81.54%
	R	50.00%	61.93%	69.32%		R	51.70%	66.48%	**74.43%**
Content (R_{lex})	MP	24.62%	17.23%	12.72%	Structure (R_{str}), Content (R_{lex}), and Quality (Q_{ehc})	MP	**31.07%**	18.62%	13.85%
	MAPK	35.00%	33.85%	33.08%		MAPK	**41.92%**	39.92%	38.64%
	TEH(65)	43(80)	49(112)	53(124)		TEH(65)	**48**(101)	**53**(121)	**56**(**135**)
	PEH	66.15%	75.38%	81.54%		PEH	73.85%	81.54%	**86.15%**
	R	45.45%	63.63%	70.45%		R	57.39%	68.75%	**76.70%**

TEH=Total exceptions handled, **PEH**=Percentage of all exceptions handled

largely distributed over *structural* and *lexical relevance* metrics compared to *exception handler quality*, and all three metrics share about 34.55% of the exceptions. More interestingly, we note that about 18% (from Fig. 3, 3.64% + 9.09% + 5.45%) exception handling cases are unique to the three metrics, which indicates that those exceptions cannot be handled or relevant code examples cannot be retrieved without considering those metrics in combination.

Table II further motivates the idea of *combined relevance* and *quality measures* with statistical evidences. It shows the results of the experiments, where we contrast among the three aspects of relevance and exception handler quality of the code examples. From Table II, we note that the different relevance aspects such as *lexical relevance* and *structural relevance* are not satisfactorily effective especially in terms of *mean average precision* and *recall*, when they are considered in isolation. For example, the approach can recommend at most 70.45% of the relevant code examples with 35.00% *mean average precision* when we consider only *lexical relevance* for ranking. On the other hand, when we consider both *structural* and *lexical relevance*, the approach can recommend with 74.43% *recall* and 37.33% *precision*. One can argue that performance improvement is not significant, which actually motivates the inclusion of another dimension in code example ranking. We consider *quality of exception handler* as the third aspect in the relevance ranking of the code examples, and we also find it promising in our experiments. When we add *handler quality* to the rest two aspects of ranking, we get a maximum *recall* of 76.70% and *mean average precision* of 41.92% by the proposed approach, and it also handles a maximum of 86.15% of all the exceptions in the dataset. While the improvement is not still too high, the combination of three aspects interestingly performs the best in terms of all performance metrics, and the results are promising. Similar findings can also be reported from Fig. 3.

F. Comparison with Existing Approaches

Even though our proposed approach shows promise in the controlled experiments above, we further wanted to see how good the approach is in terms of the literature. Thus, we compare our approach with four well known existing approaches– Barbosa et al. [13], Holmes and Murphy [19], Takuya and Masuhara [27] and Bajracharya et al. [12]. We implemented the approaches in our working environment based on the methodologies described in the paper and our prior development experience, tested with our dataset, and analyzed their performance with the same set of metrics. This section discusses the comparative study between our proposed approach and the existing approaches.

Barbosa et al. [13] developed their corpus by collecting code examples from the repositories hosted at *Eclipse Foundation Open Source Community*. They apply different preprocessing on the examples such as discarding inefficient handlers and long methods and so on, and they then apply three heuristics related to *exception type, method call* and *variable usage* for the relevance ranking. In our implementation of the approach, although we could not replicate their preprocessing steps properly, we used our example corpus as the dataset, and implemented their heuristics according to the guidelines described in the paper. We thus basically compare our proposed metrics with their proposed heuristics in terms of different experiments. Table III shows the findings of the comparative study, where we observe that their heuristic-based approach performs relatively poor in recommendation. The approach by Barbosa et al. recommends relevant code examples at most for 44.62% of the exceptions with 31.25% *recall* and 16.15% *mean average precision*, whereas our approach can recommend for 86.15% of the exceptions with 76.70% *recall* and 41.92% *mean average precision*. This clearly shows that our approach outperforms their approach. One can rationalize the lack of preprocessing for the low performance of their approach, we argue that the same limitation is also acknowledged by Barbosa et al., and this actually validates that our proposed metrics are more effective than their heuristics for the recommendation from the same corpus.

Although the remaining three approaches are not especially designed for recommending exception handling code examples, they are well known code example recommendation techniques and are closely related to our work. They also analyze either structural or lexical features from the code for recommendation, and we compare our approach against them. We implemented the existing approaches with required adjustments for the comparative study as the implementations by the authors are either unavailable or not directly applicable. The approach by Holmes and Murphy [19] uses six heuristics for code recommendation, and we find three of them are relevant for exception handling code recommendation. We thus use the three heuristics dealing with *method calls* and *variable usages* in the code. Takuya and Masuhara [27] use *cosine similarity*

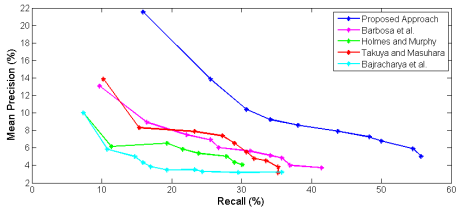

Fig. 4. Mean Precision vs. Recall Curves

in order to determine relevance between two code examples. Bajracharya et al. [12] adopt an information retrieval-based approach for code example recommendation. They extract the tokens containing different structural information from the code, and develop a *lucene index* for all the examples in the corpus. They then use a structured query containing a set of predefined parameters to collect recommendable code examples. In our implementation, we adopt a similar approach in index development involving *lucene indexer*; however, we follow a different approach for query formulation. Their query parameters [12] are not sufficient enough to request for exception handling code examples, and we use the queries (Section V-B) by our proposed approach. However, as the experiment results suggest, none of the three existing approaches perform considerably well in recommending exception handling code examples. From Table III, we note that the approach by Takuya and Masuhara *handles* a maximum of 31 (47.69%) of exceptions and recommends examples with 21.54% *mean average precision* and 30.68% *recall*, and others recommend less than 30% of all the relevant examples (i.e., *recall*), which are significantly poor compared to our results. One can argue that the comparison might not be fair due to the *handler quality metrics* in our approach. However, as shown in Table II, our approach also performs significantly better than those approaches without using those metrics. Thus, we conjecture that those approaches were not actually designed for exceptional handling code recommendation; but to the best of our knowledge they are worthy of comparison as there are no others available.

As shown in the schematic diagram in Fig. 2-(c), our approach leverages GitHub code search in dynamic corpus development. The approach thus applies ranking algorithms on a narrowed-down dataset for each exception handling case, whereas other approaches deal with a large local or remote corpus for the same. We also investigate if this additional search (i.e., GitHub search) is the sole factor behind the promising results of our approach, and conduct experiments with 4,400 code examples as the corpus for each of the exception handling cases. From Table III, we note that our approach also performs significantly well compared to the existing approaches in this case. It recommends relevant code examples for a maximum of 40 (61.54% compared to 47.69% of existing approaches) out of 65 exception handling cases with 43.75% *recall*. The *mean precision-recall* curve in Fig. 4 also shows that the proposed approach is more promising than the existing approaches in exception handling code recommendation. Given that the area under the curve denotes the performance of a system, our approach outperforms all other approaches in the experiments.

VI. Threats to Validity

In our proposed approach, we note several issues worthy of discussion. First, one might argue about the reliability of the judges for the oracle, especially because relevance checking of a code example against an exception (and its context code) is a subjective approach. In order to overcome this threat, we carefully chose the examples by consulting the best accepted practices of exception handling as well as based on our best judgment, and both authors have professional development experience (details in Section V-C).

Second, we exploit *GitHub code search API* to develop the corpus for our experiments, and our approach is subjected to strengths and weaknesses of the search feature. One might argue about the relatively smaller size of the corpus developed dynamically for each of the exception handling cases. However, we argue that those examples are actually collected from hundreds of open source repositories (about 750), and then filtered and even ranked before returning. Thus, the developed corpus was not only sufficient for our experiments but also an effective one, which is also shown by the experimental results.

Third, one might argue about the number of exceptions for the experiments. We used 65 exception handling cases for the experiments and this might not be sufficient enough to draw a generalized conclusion. However, collecting suitable cases and developing reliable oracle for them requires significant amount of time and efforts, and we covered most of the well known standard Java exceptions [9] in different cases. The corpus is also developed using examples from hundreds of code repositories hosted online. Thus we believe that the sample size is sufficient enough for a controlled experiment and to draw such a conclusion.

VII. Related Work

Exception handling is not a new topic, and there exists a good number of studies [13, 15, 16, 17, 18, 24]. Barbosa et al. [13] propose an approach to recommend exception handling code by exploiting three heuristics about structural facts in the code. The approaches by Holmes and Murphy [19], Takuya and Masuhara [27] and Bajracharya et al. [12] are well known as code recommendation techniques although they are not specialized for exception handling code. We compared our approach to all four of them and found that ours one performs significantly better than all of them. For a detailed comparison the readers are referred to Section V-F.

The other existing studies on exception handling are not directly related to code example recommendation, and thus, they were not applicable for the comparison experiments. Chang et al. [16] propose a static analysis technique that considers the exceptional control flows, and helps to discard unnecessary *try-catch* and *throw* statements. However, discarding unnecessary elements from the code may not always meet the needs of the developer in exception handling. Robillard and Murphy [24] propose another static analysis approach that identifies different possible exceptional flows in the application program and helps the developer to understand

TABLE III
COMPARISON WITH EXISTING APPROACHES

Recommender	Metric	Top 5	Top 10	Top 15	Recommender	Metric	Top 5	Top 10	Top 15
Barbosa et al. [13]	MP	8.92%	6.92%	5.64%	Proposed Approach (without GitHub search) Structure(R_{str}) only	MP	**13.54%**	8.77%	7.18%
	MAPK	**16.15%**	14.69%	13.72%		MAPK	**21.80%**	19.87%	18.85%
	TEH(65)	18(29)	25(45)	29(55)		TEH(65)	**30**(44)	**33**(57)	**37**(70)
	PEH	27.69%	38.46%	**44.62%**		PEH	46.15%	50.77%	56.92%
	R	16.47%	25.57%	**31.25%**		R	**25.00%**	**32.38%**	**39.77%**
Holmes and Murphy [19]	MP	6.15%	5.85%	5.03%	Proposed Approach (without GitHub search) Structure(R_{str}), Content(R_{lex}), and Quality(Q_{ehc})	MP	**13.85%**	9.23%	7.90%
	MAPK	4.62%	2.31%	2.31%		MAPK	**30.64%**	27.44%	25.90%
	TEH(65)	16(20)	25(38)	31(49)		TEH(65)	**31**(45)	**34**(60)	**40**(77)
	PEH	24.62%	38.46%	**47.69%**		PEH	**47.69%**	52.31%	**61.54%**
	R	11.36%	21.59%	**27.84%**		R	**25.56%**	**34.09%**	**43.75%**
Takuya and Masuhara [27]	MP	8.31%	7.38%	5.54%	Proposed Approach (with GitHub search) Structure(R_{str}) and Content(R_{lex})	MP	**27.99%**	17.99%	13.44%
	MAPK	**21.54%**	20.51%	19.74%		MAPK	**43.07%**	38.69%	37.33%
	TEH(65)	22(27)	31(48)	31(54)		TEH(65)	45(91)	49(117)	**53**(131)
	PEH	33.85%	47.69%	**47.69%**		PEH	69.23%	75.38%	**81.54%**
	R	15.34%	27.27%	**30.68%**		R	**51.70%**	**66.48%**	**74.43%**
Bajracharya et al. [12]	MP	5.85%	4.31%	3.49%	Proposed Approach (with GitHub search) Structure(R_{str}), Content(R_{lex}), and Quality(Q_{ehc})	MP	**31.07%**	18.62%	13.85%
	MAPK	8.46%	7.95%	6.41%		MAPK	**41.92%**	39.92%	38.64%
	TEH(65)	12(19)	18(28)	20(34)		TEH(65)	**48**(101)	**53**(121)	**56**(135)
	PEH	18.46%	27.69%	**30.77%**		PEH	73.85%	81.54%	**86.15%**
	R	10.80%	15.91%	**19.32%**		R	**57.39%**	**68.75%**	**76.70%**

TEH=Total exceptions handled, **PEH**=Percentage of all exceptions handled

and improve the exception handling structures of the system. However, it returns thousands of possible control flow paths for an exception, and that information is not easy to use in practical sense [13]. Thus, in general, the static analysis-based techniques provide limited support for instant exception handling from the first place, and they often assume that handling is already done somehow and the handler code is there [13]. Garcia et al. [17] conduct an empirical study on the exception handling mechanisms available in different object-oriented programming languages, and propose a new exception handling structure that considers 10 important aspects related to handling. Shah et al. [26] propose a visualization approach that visualizes the exception handling structures in the large software systems for better understanding of how the system works. Thus while other studies provide useful insights into the control flows, handling structures through static analysis, field studies, empirical studies and visualization, our proposed approach provides readily available and relevant working code examples by exploiting context code in the IDE, which can be easily leveraged for exception handling.

VIII. CONCLUSION & FUTURE WORKS

To summarize, we propose a context-aware code recommender that recommends exception handling code examples against the code under development (i.e., context code) in the IDE. We consider three aspects–*structure, content* and *handler quality* of the candidate code examples for relevance ranking. Experiments with 65 exceptions (and their context code) and 4,400 code examples as well as comparisons with four existing approaches show that our approach is highly promising. While our experiments show that the general-purpose code recommendation approaches are not satisfactorily applicable for the recommendation of exception handling code, in this paper, our technical contribution lies in proposing a graph-based approach for structural relevance estimation, introducing handler quality dimension in relevance ranking, and developing an Eclipse plugin. In future, we plan to conduct a user study with prospective participants.

REFERENCES

[1] Cosine Similarity. URL http://en.wikipedia.org/wiki/Cosine_similarity.
[2] GitHub Code Search. URL http://developer.github.com/v3/search/.
[3] Graph Matching. URL http://en.wikipedia.org/wiki/Matching_(graph_theory).
[4] Exception Handling Principles. URL http://howtodoinjava.com/2013/04/04/java-exception-handling-best-practices.
[5] Best Practices for Exception Handling. URL https://www.ibm.com/developerworks/library/j-ejbexcept.
[6] Javaparser-Java 1.5 Parser and AST. URL http://code.google.com/p/javaparser.
[7] Logistic Regression. URL http://en.wikipedia.org/wiki/Logistic_regression.
[8] Pastebin. URL http://pastebin.com.
[9] SurfExample Portal. URL http://www.usask.ca/~mor543/surfexample.
[10] Readability Library. URL http://www.arrestedcomputing.com/readability.
[11] Weka. URL http://www.cs.waikato.ac.nz/ml/weka/.
[12] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An Internet-Scale Software Repository. In *Proc. SUITE*, pages 1–4, 2009.
[13] E. A. Barbosa, A. Garcia, and M. Mezini. Heuristic Strategies for Recommendation of Exception Handling Code. In *Proc. SBES*, pages 171–180, 2012.
[14] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *TSE*, 36(4):546–558, 2010.
[15] B. Cabral and P. Marques. Exception Handling: A Field Study in Java and .NET. In *Proc. ECOOP*, pages 151–175, 2007.
[16] B. M. Chang, J. W. Jo, K. Yi, and K. M. Choe. Interprocedural Exception Analysis for Java. In *Proc. SAC*, pages 620–625, 2001.
[17] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *JSS*, 59(2):197–222, 2001.
[18] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Commun. ACM*, 18(12):683–696, 1975.
[19] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*, pages 117–125, 2005.
[20] B. T. S. Kumar and J. N. Prakash. Precision and Relative Recall of Search Engines: A Comparative Study of Google and Yahoo. *J. Lib. and Info. Mgmt.*, 38(1):124–137, 2009.
[21] C. Le Goues and W. Weimer. Measuring Code Quality to Improve Specification Mining. *TSE*, 38(1):175–190, 2012.
[22] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proc. ESEC/FSE*, pages 383–392, 2009.
[23] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.
[24] M. P. Robillard and G. C. Murphy. Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. *TOSEM*, 12(2):191–221, 2003.
[25] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. ICPC*, pages 172–181, 2008.
[26] H. Shah, C. Görg, and M. J. Harrold. Visualization of Exception Handling Constructs to Support Program Understanding. In *Proc. SoftVis*, pages 19–28, 2008.
[27] W. Takuya and H. Masuhara. A Spontaneous Code Recommendation Tool Based on Associative Search. In *Proc. SUITE*, pages 17–20, 2011.
[28] T. Usmani, D. Pant, and A. K. Bhatt. A Comparative Study of Google and Bing Search Engines in Context of Precision and Relative Recall Parameter. *J. CSE*, 4 (1):21–34, 2012.