Exploiting Context in Dealing with Programming Errors and Exceptions in the IDE

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Mohammad Masudur Rahman

©Mohammad Masudur Rahman, August/2014. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Studies show that software developers spend about 19% of their development time in web surfing. While collecting necessary information using traditional web search, they face several practical challenges. First, it does not consider context (i.e., surroundings, circumstances) of the programming problems during search unless the developers do so in search query formulation, and forces the developers to frequently switch between their working environment (e.g., IDE) and the web browser. Second, technical details (e.g., stack trace) of an encountered exception often contain a lot of information, and they cannot be directly used as a search query given that the traditional search engines do not support long queries. Third, traditional search generally returns hundreds of search results, and the developers need to manually analyze the result pages one by one in order to extract a working solution. Both manual analysis of a page for content relevant to the encountered exception (and its context) and working an appropriate solution out are non-trivial tasks. Traditional code search engines share the same set of limitations of the web search ones, and they also do not help much in collecting the code examples that can be used for handling the encountered exceptions.

In this thesis, we present a context-aware and IDE-based approach that helps one overcome those four challenges above. In our first study, we propose and evaluate a context-aware meta search engine for programming errors and exceptions. The meta search collects results for any encountered exception in the IDE from three popular search engines–*Google*, *Bing* and *Yahoo* and one programming Q & A site–*StackOverflow*, refines and ranks the results against the detailed context of the encountered exception, and then recommends them within the IDE. From this study, we not only explore the potential of the context-aware and meta search based approach but also realize the significance of appropriate search queries in searching for programming solutions. In the second study, we propose and evaluate an automated query recommendation approach that exploits the technical details of an encountered exception, and recommends a ranked list of search queries. We found the recommended queries quite promising and comparable to the queries suggested by experts. We also note that the support for the developers can be further complemented by post-search content analysis. In the third study, we propose and evaluate an IDE-based context-aware content recommendation approach that identifies and recommends sections of a web page that are relevant to the encountered exception in the IDE. The idea is to reduce the cognitive effort of the developers in searching for content of interest (i.e., relevance) in the page, and we found the approach quite effective through extensive experiments and a limited user study. In our fourth study, we propose and evaluate a context-aware code search engine that collects code examples from a number of code repositories of *GitHub*, and the examples contain high quality handlers for the exception of interest. We validate the performance of each of our proposed approaches against existing relevant literature and also through several mini user studies. Finally, in order to further validate the applicability of our approaches, we integrate them into an Eclipse plug in prototype–*ExcClipse*. We then conduct a task-oriented user study with six participants, and report the findings which are significantly promising.

Acknowledgements

At first, I thank the Almighty, the most gracious and the most merciful, who granted me the capability to carry out this work. Then I would like to express my heartiest gratitude to my supervisor Dr. Chanchal K. Roy for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without his support this work would have been impossible.

I would like to thank Dr. Kevin Stanley, Dr. Kevin A. Schneider and Dr. Rama Gokaraju for their willingness to take part in the advisement and evaluation of my thesis work. I would also like to thank them for their valuable time, suggestions and insights.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Md. Saidur Rahman, Manishankar Mondal, Muhammad Asaduzzaman, Farouq Al-Omari, Minhaz Fahim Zibran, Md. Sharif Uddin, Mohammad Asif Ashraf Khan, Khalid Billah, Jeffrey Svajlenko, and Judith Islam.

I am grateful to the University of Saskatchewan and its Department of Computer Science for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I thank all the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach at this stage. In particular I would like to thank Richard K. Lomotey, Sayooran Nagulendra, Dylan Knowles and Gwen Lancaster, Shakiba Jalal, and Heather Webb.

I would like to convey my love and gratitude to my beloved wife, Shamima Yeasmin, who has brought a new meaning to my life. She always stayed with me in ease and hardship, inspired me constantly, and helped me with ideas and suggestions in this work.

I express my heartiest gratitude to my mother Mrs. Morium Begum, and my father Md. Sadiqur Rahman who are the architects of my life. Their endless sacrifice, unconditional love and constant well wishes have made me reach at this stage of my life. I would also like to thank my mother-in-law Mrs. Rezia Khatun and father-in-law Md. Shamsul Islam for their constant well wishes and inspirations in this thesis work. My brothers– Asad, Mamun and Sayed, and in-laws– Masum, Mamun, Maruf, Rabeya and Farzana have always inspired me in completing my thesis work, and I thank all of them.

I dedicate this thesis to my mother Mrs. Morium Begum and my father Md. Sadiqur Rahman whose inspirations help me to accomplish every step of my life.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables viii

List of Figures ix

List of Abbreviations x

1 Introduction 1
- 1.1 Motivation 1
  - 1.1.1 Problem Statement 2
  - 1.1.2 Our contribution 3
- 1.2 Related Publications 4
- 1.3 Outline of the Thesis 4

2 Background 6
- 2.1 Cosine Similarity 6
- 2.2 Degree Of Interest (DOI) 7
- 2.3 Document Object Model (DOM) 8
- 2.4 Static Relationship and Data Dependency 8
- 2.5 Stack Trace Token Graph 9
- 2.6 Graph Matching 10
- 2.7 Logistic Regression 11
- 2.8 Page Rank Algorithm 11
- 2.9 Summary 12

3 An IDE-Based Context-Aware Meta Search Engine 13
- 3.1 Introduction 13
- 3.2 Proposed Approach for IDE-Based Context-Aware Meta Search 14
  - 3.2.1 Proposed System Model 14
  - 3.2.2 Proposed Metrics 16
  - 3.2.3 Result Scores Calculation 21
- 3.3 Experimental Design, Results and Validation 22
  - 3.3.1 Dataset Preparation 22
  - 3.3.2 Investigation with Eclipse IDE 22
  - 3.3.3 Search Query Formulation 23
  - 3.3.4 Performance Metrics 23
  - 3.3.5 Experimental Results on Proposed Approach 24
  - 3.3.6 Comparison with Existing Approaches 25
  - 3.3.7 Comparison with Existing Search Engines 27
- 3.4 Threats to Validity 30
- 3.5 Related Work 30
- 3.6 Summary 31

4 Context-Aware Search Query Recommendation 32
4.1 Introduction . . . 32
4.2 Motivating Example . . . 34
4.3 Proposed Approach . . . 35
4.3.1 Methodology . . . 36
4.3.2 Proposed Metrics . . . 37
4.3.3 Token Score Calculation . . . 39
4.3.4 Query Ranking & Recommendation . . . 39
4.4 Experimental Design, Results and Validation . . . 40
4.4.1 Dataset and Tools . . . 40
4.4.2 Existing Query Recommender . . . 40
4.4.3 Performance Metrics . . . 41
4.4.4 Experimental Results . . . 42
4.4.5 Comparison with Existing Approaches . . . 43
4.4.6 Conducted User Study . . . 46
4.5 Threats to Validity . . . 47
4.6 Related Work . . . 48
4.7 Summary . . . 49

5 Context-Aware Content Suggestion from a Web Page 50
5.1 Introduction . . . 50
5.2 Proposed Approach . . . 53
5.2.1 Working Modules . . . 53
5.2.2 Proposed Metrics . . . 55
5.2.3 Content Score Calculation . . . 58
5.2.4 Metric Weight Estimation . . . 59
5.2.5 Document Content Extraction . . . 59
5.3 Experimental Design, Results and Validation . . . 60
5.3.1 Dataset Preparation . . . 60
5.3.2 Traditional Web Page Content Recommendation . . . 61
5.3.3 Exception Context Representation . . . 61
5.3.4 Performance Metrics . . . 62
5.3.5 Experimental Results . . . 62
5.3.6 Comparison with Existing approaches . . . 64
5.4 Threats to Validity . . . 67
5.5 Related Work . . . 67
5.6 Summary . . . 68

6 IDE-Based Context-Aware Exception Handling Code Example Recommendation 70
6.1 Introduction . . . 70
6.2 Motivating Example . . . 72
6.3 Proposed Approach . . . 75
6.3.1 Working Modules . . . 75
6.3.2 Proposed Metrics . . . 75
6.3.3 Result Scores and Ranking . . . 79
6.3.4 Metric Weight Estimation . . . 79
6.4 Experimental Design, Results and Validation . . . 80
6.4.1 Dataset Preparation . . . 80
6.4.2 Search Query Formulation . . . 81
6.4.3 Exception Oracle Development . . . 81
6.4.4 Performance Metrics . . . 81
6.4.5 Experimental Results . . . 82
6.4.6 Comparison with Existing Approaches . . . 86
6.5 Threats to Validity . . . 88

6.6 Related Work . . . 88
6.7 Summary . . . 89

7 A Comparison between Proposed Methods and Traditional Ones: A Task-Based User Study 90
7.1 ExcClipse . . . 90
7.1.1 ExcClipse Web Search . . . 91
7.1.2 ExcClipse Code Example Search . . . 93
7.2 Design of User Study . . . 95
7.2.1 Tasks Design . . . 95
7.2.2 Exception Test Cases . . . 96
7.2.3 Study Participants . . . 96
7.2.4 Study Data Collection . . . 96
7.3 Running the Study Session . . . 97
7.3.1 Training Phase . . . 97
7.3.2 Execution Phase . . . 97
7.3.3 Evaluation Phase . . . 97
7.4 Result Analysis and Discussions . . . 98
7.4.1 Evaluation Metrics . . . 98
7.4.2 Motivating Factors for ExcClipse . . . 99
7.4.3 Comparison of ExcClipse with Traditional Search Providers . . . 100
7.4.4 Qualitative Suggestions from Participants . . . 106
7.5 Threats to Validity . . . 106
7.6 Summary . . . 107

8 Conclusion 108
8.1 Concluding Remarks . . . 108
8.2 Future Work . . . 109

A User Study Test Cases 116
A.1 Exception Test Case 1 (EC_1) . . . 116
A.1.1 Stack Trace . . . 116
A.1.2 Context Code . . . 116
A.2 Exception Test Case 2 (EC_2) . . . 116
A.2.1 Stack Trace . . . 116
A.2.2 Context Code . . . 116
A.3 Exception Test Case 3 (EC_3) . . . 117
A.3.1 Stack Trace . . . 117
A.3.2 Context Code . . . 117
A.4 Exception Test Case 4 (EC_4) . . . 117
A.4.1 Stack Trace . . . 117
A.4.2 Context Code . . . 117

B User Study Data Collection Techniques 118
B.1 Observation Checklist . . . 118
B.1.1 Observation Checklist for Traditional Search . . . 118
B.1.2 Observation Checklist for ExcClipse . . . 119
B.2 Questionnaire . . . 120

List of Tables

3.1 Results of Experiments on Two Working Modes of Proposed Approach 24
3.2 Experimental Results for Different Ranking Aspects 26
3.3 Results of Experiments on Multiple Sets 27
3.4 Comparison with Existing Approaches and Search Engines 28
3.5 Impact of Context, and Common and Unique Results from Search Engines 29

4.1 Pyramid Score Calculation 41
4.2 Results of Experiments on Different Aspects for Token Importance 43
4.3 Results of Experiments on Different Ranked Queries 44
4.4 Comparison with Queries from Traditional and Existing Approaches 44
4.5 Comparison with User Provided Queries for Exceptions 47

5.1 Dataset Statistics 61
5.3 Experimental Results for Different Aspects of Page Content 63
5.2 Results of Experiments on Main (i.e., noise-free) and Relevant Content 63
5.4 Comparison with Existing Approaches 65

6.1 Experimental Results 82
6.2 Experimental Results on Different Aspects of Code 83
6.3 Comparison with Existing Approaches 85

7.1 Study Session Plan 98
7.2 Study Execution Sessions 98
7.3 Severity, Frequency and Relevance Scales 99
7.4 Participants' Ratings for Motivating Factors behind ExcClipse 99
7.5 Tool Features for Evaluation 100
7.6 Rating of Tool Features by Participants 102
7.7 Overall Rating Difference among Participants 105
7.8 Observed Responses and Feedback from Participants 105

List of Figures

2.1 DOM Tree of the Example HTML Page in Listing 2.1 7
2.2 Static Relationships and Data Dependencies in Listing 2.2 9
2.3 Stack Trace Token Graph of Listing 2.3 10

3.1 Schematic Diagram of Proposed Approach (SurfClipse) 15

4.1 Stack Trace Token Graph of Stack Trace in Listing 4.2 35
4.2 Schematic Diagram of Proposed Approach (QueryClipse) 36

5.1 An Example Relevant Section for the Exception in Listing 5.2 53
5.2 Relevant Section(s) in the Webpage 54
5.3 Schematic Diagram of Proposed Approach (ContentSuggest) 55

6.1 Schematic Diagram of Proposed Approach (SurfExample) 74
6.2 Result Distribution over Different Metrics 83
6.3 Mean Precision vs. Recall Curves 87
6.4 Mean Average Precision vs. Recall Curves 87

7.1 ExcClipse Web Search 91
7.2 Web Search Query Recommendation (Interactive Mode) 92
7.3 Context-Menu Based Search (Interactive Mode) 93
7.4 Web Page Browser 93
7.5 ExcClipse Code Example Search 94
7.6 Context-Menu Based Code Search 95
7.7 Code Search Query Recommendation 95
7.8 Ratings of Web Search Features 101
7.9 Ratings of Code Search Features 101
7.10 Ratings of Non-functional Features 103
7.11 Overall Ratings from Participants (Difference is Significant, U=0, p=0.0051) 104
7.12 Agreement among Ratings from Participants (i.e., Edges refer to Agreements) 104

List of Abbreviations

ACCB	Adaptive Code Content Blurring
AHA	Average Handler Actions
AOM	API Object Match
API	Application Programming Interface
CD	Code Density
COTS	Commercial Off-The-Shelf
CR	Code Relevance
CSE	Code Search Engine
CTD	Content Density
CTR	Content Relevance
CTS	Content Score
DDM	Data Dependency Match
DOM	Document Object Model
DSC	Document Slope Curves
DSO	Data from StackOverflow
EOF	End Of File
FAM	Field Access Match
FAQ	Field Access in Query
HCR	Handler to Code Ratio
IDE	Integrated Development Environment
LCS	Longest Common Subsequence
LD	Link Density
MAPK	Mean Average Precision at K
MFFP	Mean First False-Positive Position
MF	Mean $F_1 - measure$
MIM	Method Invocation Match
MIQ	Method Invocation in Query
MP	Mean Precision
MRR	Mean Reciprocal Rank
MR	Mean Recall
PTCS	Percentage of Test Cases Solved
PEH	Percentage of Exceptions Handled
RA	Readability
R	Recall
SO	StackOverflow
TCCB	Token-based Code Content Blurring
TCS	Test Cases Solved
TD	Text Density
TEF	Total Exceptions Fixed
TEH	Total Exceptions Handled
TF-IDF	Term Frequency-Inverse Document Frequency
TR	Text Relevance
TTF	Trace Token Frequency
TTR	Trace Token Rank

Chapter 1

Introduction

1.1 Motivation

Studies show that up to 85%–90% of global cost of a software system is spent in its maintenance [70, 78]. During the development and the maintenance of a software system, software developers face different programming challenges, and one of the major challenges is– dealing with programming errors and exceptions. Existing IDEs (e.g., Eclipse, Visual Studio, NetBeans) are equipped with various debugging supports for the encountered errors and exceptions, and a developer may get useful clues about the solution from the *stack trace*[1] of an exception reported by an IDE. However, neither the tracing information nor the *context code*[2] in the IDE often help enough in resolving the exceptions, especially when the developer lacks necessary skills (i.e., novice developer) or the encountered programming errors or exceptions are relatively unfamiliar (i.e., new API exceptions). Thus the developers often look into web and search for more helpful and up-to-date information. According to the study of Brandt et al. [35], developers spend about 19% of their development (i.e., programming) time in web surfing.

While collecting information using traditional web search, developers face several practical challenges. First, the traditional search does not consider *context* (i.e., surroundings, circumstances) of the programming problems during search unless the developers prepare queries good enough by analyzing their *context*, which is a non-trivial task. It also forces the developers to leave their working environment (e.g., IDE) and look for the search results in the web browser. The keyword-based traditional search often does not help much in problem solving, and the frequent switching between the IDE and the web browser is both distracting and time-consuming. Second, the stack trace of an encountered exception reported by the IDE generally contains a lot of information, and it cannot be directly used as a search query given that the traditional search engines such as *Google, Bing* or *Yahoo* do not support long queries. On the other hand, the technical error message (i.e., generally found in the first line of the stack trace) often contains a very limited information (e.g., an exception name only) which is not sufficient enough for a search query. Thus the developers often face difficulties in choosing a suitable search query for the encountered error or exception. Third, traditional search generally returns hundreds of result pages, and developers manually analyze them one by one in order

[1] A report of the active stack frames at a certain point in time during the execution of a program
[2] A segment of the code that triggers the exception

to extract a working solution. Both manual checking of a page for content relevant to the exception (and its context), and working an appropriate solution out are non-trivial tasks. They are even more complex and time-consuming with the bulk of irrelevant and noisy (e.g., advertisements) sections in the pages. Most of the modern programming languages (e.g., object-oriented, functional) provide exception handling features that enable one to deal with the programming errors and exceptions. However, studies show that the developers either misuse those features [77] or use them ineffectively [38] during software development. They even consider effective exception handling as either a daunting or a counter-productive task. One way to support them in this regard is to recommend code examples containing quality handlers for the exception of interest. Existing code search engines such as *Ohloh* [18], *Krugle* [16] or *GitHub Code Search* [7] suffer from the same set of limitations of the web search ones. They also do not consider the context of the programming exception to be handled in the IDE, and thus do not help much in finding relevant code examples that can be used for exception handling.

1.1.1 Problem Statement

A number of existing studies are conducted to address the above challenges with traditional web search [43, 50, 51, 68, 69, 70, 79] and code search [32, 33, 55, 80]. Unfortunately, none of them except a few [33, 43] are targeted for dealing with programming errors and exceptions. There exist other studies focusing on programming errors and exceptions, that recommend simplified error messages [62], relevant bug reports [52], and exception solving tips [54]. However, they do not involve ones in the web search or in the code example search. An error or an exception in the IDE is generally associated with the technical details comprising of an error message and a stack trace, and the exception is triggered from a certain programming context. A web search or a code search that retrieves an effective solution or a quality handler for the exception should take the technical details and the programming context into account. Most of those existing approaches are not IDE-based, and thus they ignore the technical details and the context code in the IDE and rely entirely on the user provided search queries. Thus if the queries are not prepared carefully or fail to capture the context of the problems, they may not return useful results. The existing IDE-based approaches apply the technical details (e.g., stack trace) and the code under development either from different perspectives (i.e., not targeted for exceptions) [69, 70] or in a limited fashion (i.e., analyzes stack trace only) [43]. Thus these approaches are either non-applicable or not very effective in the search related to programming errors and exceptions. In this thesis, we attempt to address the above four challenges through an integrated and IDE-based solution that leverages the available information in its disposal. Thus the research problem we attempt to solve is– the effective and diversified use of the technical details and the programming context of an error or exception in different search-related activities. We analyze the research problem and formulate the following research questions, which we attempt to answer in this thesis:

- Is a context-aware meta search more effective than a keyword-based traditional web search for programming errors and exceptions?

- Are context-aware search queries more effective than traditional or user provided queries for the search with traditional web search engines?
- How the technical details and the context of an exception can be used in recommending relevant sections from a web page?
- Is a context-aware code search more effective than a traditional code search or an existing approach from the literature for collecting exception handling code examples?

1.1.2 Our contribution

In this thesis, we propose and evaluate four novel approaches that support developers in different search-related activities associated with programming errors and exceptions. We consider the error message, stack trace and programming context (i.e., context code) in the IDE as the *context* of an encountered exception, and leverage them in the recommendation of different items such as search queries, relevant web pages, relevant sections of a given web page, and relevant code examples for exception handling. In the first study, we propose *SurfClipse*, an IDE-based context-aware meta search engine, that collects search results from three popular search engines–*Google, Bing* and *Yahoo* and one programming Q & A site–*StackOverflow*, for an encountered exception, refines them against the *context* of the exception, and then returns the ranked results. The proposed engine outperforms the traditional search engines and similar existing approaches in terms of different performance metrics. In the second study, we propose *QueryClipse* that exploits the *context* of an exception and recommends context-aware search queries for the exception. The recommended queries are found to be more effective compared to traditional search queries or the queries by existing approaches, and found comparable to the queries suggested by experts. In the third study, we focus on the use of such *context* in post-search content analysis in order to help developers collect relevant information with reduced effort. In this study, our proposed approach, *ContentSuggest*, exploits the *context* of an exception, and recommends such sections of a web page that are relevant to the exception. In the last study, we propose *SurfExamle*, a context-aware code search engine, that collects exception handling code examples from *Github* repositories for an exception, refines them against the *context* of the target exception in the IDE, and then returns the ranked code examples for exception handling. All four proposed approaches are evaluated extensively, and their performance is also validated against existing relevant approaches. Finally, in order to validate the applicability of those approaches in real world software development, we incorporate them into an Eclipse plug in, *ExcClipse*, and conduct a task-oriented user study with six participants. In this study, each of the participants performs four search-related tasks associated with four programming exceptions using both *ExcClipse* and traditional search engines, where they compare our prototype with traditional counterpart for different technical features and supports. The study shows that our proposed approaches are promising and our prototype is significantly preferred to traditional means for the provided features and supports.

1.2 Related Publications

Several parts of this thesis are published and accepted in different conferences, and we provide the list of publications here. In each of the papers, I am the primary author, and all the studies are conducted by me under the supervision of Dr. Chanchal K. Roy. While I wrote the papers, the co-authors took part in advising, editing, and reviewing the papers.

- M. Masudur Rahman, S. Yeasmin, and C.K. Roy, "An IDE-Based Context-Aware Meta Search Engine", *Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 467-471

- M. Masudur Rahman, S. Yeasmin, and C.K. Roy, "Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions", *CSMR/WCRE Software Evolution Week (SEW)*, 2014, pp. 194-203.

- M. Masudur Rahman and C.K. Roy, "SurfClipse: Context-Aware Meta Search in the IDE", *International Conference on Software Maintenance and Evolution (ICSME)*, 4 pp., 2014 (to appear).

- M. Masudur Rahman and C.K. Roy, "On the Use of Context in Recommending Exception Handling Code Examples", *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 10 pp., 2014 (to appear).

1.3 Outline of the Thesis

The thesis contains eight chapters in total. In order to deal with programming errors and exceptions efficiently, we conduct four independent but interrelated studies followed by a user study, and this section outlines different chapters of the thesis.

- Chapter 2 discusses several background concepts of this thesis such as static relation and data dependency graphs, text or graph similarity matching algorithms, PageRank algorithm and so on.

- Chapter 3 discusses the first study that proposes *SurfClipse*, a context-aware meta search engine. It analyzes the context details (e.g., stack trace, context code) of an encountered exception in the IDE, and recommends relevant web pages by exploiting multiple search engines.

- Chapter 4 studies design and effectiveness of context-aware search queries in web search, and proposes an Eclipse plugin, *QueryClipse*.

- Chapter 5 focuses on *ContentSuggest* that exploits the technical details (e.g., stack trace) and programming context (e.g., context code) of an encountered exception, and recommends such sections from a web page that are both noise-free and relevant to the exception in the IDE.

- Chapter 6 discusses our fourth study that proposes *SurfExample*, a context-aware code search engine, for exception-handling code examples. It analyzes the code under development in the IDE, and recommends relevant code examples from GitHub repositories for exception handling.
- Chapter 7 discusses detailed design and findings of the conducted user study.
- Chapter 8 concludes the thesis with a list of directions for future works.

Chapter 2

Background

In this chapter, we introduce the required terminologies and concepts to follow the remaining of the thesis. Section 2.1 defines cosine similarity, a text similarity matching algorithm, Section 2.2 defines degree of interest, a heuristic measure associated with stack trace of an exception, and Section 2.3 illustrates Document Object Model (DOM). Section 2.4 and Section 2.5 discuss static relationship and data dependency graph as well as stack trace token graph respectively, and Section 2.6 focuses on graph-matching. Section 2.7 defines logistic regression, Section 2.8 explains PageRank algorithm and finally, Section 2.9 summarizes the chapter.

2.1 Cosine Similarity

Cosine Similarity[1] is a measure that indicates the orientation between two vector spaces with different number of dimensions. It is frequently used in information retrieval in order to estimate similarity between two text documents, where each distinct term within the documents is considered as a dimension and each document is considered as a vector of such terms. In our research, we often use cosine similarity measure to determine relevance between two stack traces, two code segments [80] or a search query and the discussion texts of a candidate web page. We consider each of the two items of interest (e.g., search query and text block in the web page) as a *bag of words*[2], remove the *stop words* (i.e., insignificant words in a sentence), and then perform *stemming* (i.e., extracting the root of a word using *Porter Algorithm*[3]) on each term (especially required for natural language texts) which provides a normalized form of the term. We prepare a combined vector of normalized terms, C, from the two items, and then calculate cosine similarity (S_{cos}) applying the following equation:

$$S_{cos} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.1}$$

Here, A_i represents the frequency of ith term from C in vector A (i.e., search query), and B_i represents that frequency in vector B (i.e., text block in the web page). This measure values from zero (i.e., complete lexical dissimilarity) to one (i.e., complete lexical similarity), and helps to estimate the lexical relevance between the search query and the web page.

[1] http://en.wikipedia.org/wiki/Cosine_similarity
[2] A collection of words with no fixed order
[3] http://ir.dcs.gla.ac.uk/resources/linguistic_utils/porter.java

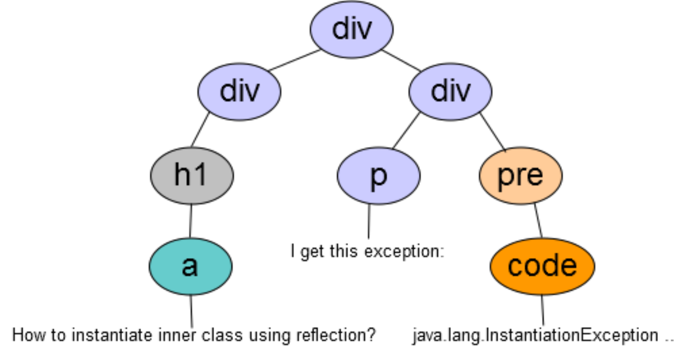

Figure 2.1: DOM Tree of the Example HTML Page in Listing 2.1

Listing 2.1: An Example HTML Segment (adapted from [13])

```
<div id="content">
<div id="question—header">
<h1 itemprop="name">
<a>How to instantiate inner class using
reflection?</a></h1></div>
<div class="post—text" itemprop="description">
<p>I get this exception:</p>
<pre class="lang—java prettyprint prettyprinted">
<code>java.lang.InstantiationException ..</code>
</pre></div></div>
```

2.2 Degree Of Interest (DOI)

The stack trace of an encountered exception comprises of an error message and a list of method call references. Cordeiro et al. [43] propose *Degree Of Interest (DOI)*, which is a measure associated with the method call references of a stack trace. They consider the measure as an *estimate of proximity* of a call reference (and its terms) to the location of the target exception in the code. In our research, we leverage the measure in determining not only the relevance between two stack traces but also the appropriateness of a term of the stack trace in a search query for the corresponding exception. Suppose, a stack trace has N method call references, then *degree of interest (DOI)* measure, S_{doi}, for each call reference can be calculated as follows:

$$S_{doi} = 1 - \frac{n_i - 1}{N} \tag{2.2}$$

Here, n_i represents the position of the reference in the stack trace. The measure values from zero to one, where zero represents that the reference is the the most distant one from the location of exception, and one means that it is the most likely reference that generates the exception. Thus the top-most reference in the trace information is of the highest interest.

2.3 Document Object Model (DOM)

It is a cross-platform and language independent convention to represent the content (i.e., objects) of an HTML or an XML document [6]. In this model, a document is considered as a tree, where each of the tags is represented as an *inner node* and textual or graphical elements are represented as *leaf nodes*. For example, the HTML code segment in Listing 2.1 shows the title and body of a programming question posted on StackOverflow Q & A site, and Fig. 2.1 shows the corresponding DOM tree. In our research, we use *Jsoup* [15], a popular Java library, in order to parse and analyze the DOM tree of HTML web pages.

2.4 Static Relationship and Data Dependency

Nguyen et al. [67] 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 our research, we exploit the *static relationships* between API objects and their properties (e.g., methods, fields and constructors) as well as the *data dependencies* among different objects in the code in order to determine the *structural relevance* between two code segments. Listing 2.2 shows a code example that reads from and writes data to a network using a *Socket* object. Fig. 2.2 shows the static relationships (i.e., green coloured solid edges) and the data dependencies (i.e., magenta coloured dashed edges) in the code example (Listing 2.2) that uses four API classes– *Socket, PrintWriter, InputStreamReader* and *BufferedReader*. We note that *PrintWriter* and *InputStreamReader* objects access *Socket* object, and *BufferedReader* object accepts an instance of *InputStreamReader* class in its constructor, and the dependencies are identified as dashed edges. On the other hand, object to property (e.g., method) static relationships are represented as green coloured solid edges.

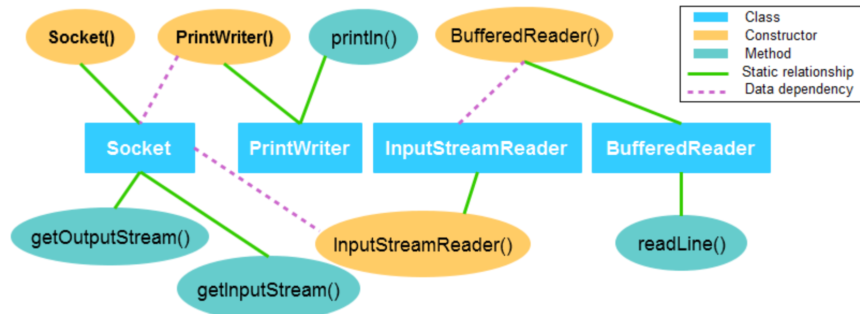

Figure 2.2: Static Relationships and Data Dependencies in Listing 2.2

Listing 2.2: An Example Code Segment (adapted from [2])

```
try {
  Socket socket = new Socket(hostName, 15432);
  PrintWriter out = new PrintWriter(socket.getOutputStream(),true);
  BufferedReader input = new BufferedReader(new InputStreamReader(socket.
      getInputStream()));
  //sending and receiving data from the network
  out.println(Integer.toString(menuSelection));
  String outputString;
  while (((outputString = input.readLine()) != null) &&
  (!outputString.equals("END_MESSAGE"))) {
      //process the output
  } }
  catch (Exception e){} // generic exception handler
```

2.5 Stack Trace Token Graph

We adapt the graph-based approach of Nguyen et al. (Section 2.4) for the stack trace of an exception, and encode the implied static relationships and the call dependencies (i.e., temporal usage order) found in the trace information into a graph, hereby we call it *stack trace token graph*. Listing 2.3 shows an example stack trace containing such relationships and dependencies, and it involves four Java classes– *ObjectInputStream, PeekInputStream, DataInputStream* and *HighScores* and five methods–*main, init, readStreamHeader, readShort* and *readFully*. While the Java objects are statically related to their corresponding methods, and we also note that the program control flows from *HighScores.main()* method to *PeekInputStream.readFully()* method through a series of intermediate method invocations. The corresponding token graph for the stack trace in Fig. 2.3 visualizes those static relationships and call dependencies among different program tokens (e.g., class name and method name). In the graph, each of the class tokens and method tokens are represented as vertices, and both *class-to-method* relationships (e.g., *ObjectInputStream*-to-*readStreamHeader*)

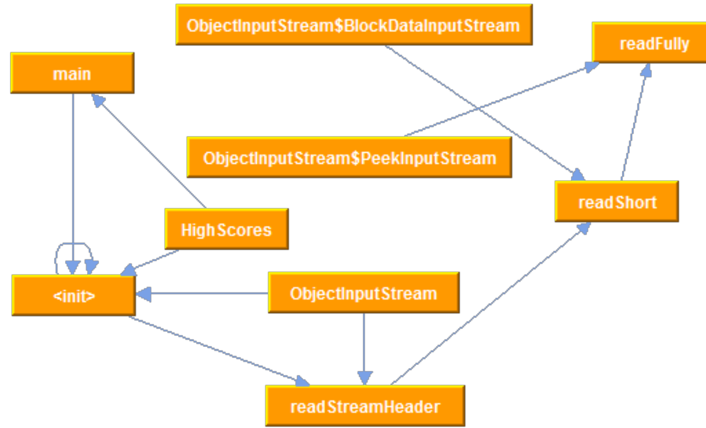

Figure 2.3: Stack Trace Token Graph of Listing 2.3

and call dependencies or control flows (e.g., *readShort-to-readFully*) are represented as directed connecting edges among those vertices. The graph provides a mean to explore the connectivity (i.e., importance) of a token in the token network developed from the stack trace, and we exploit such information in order to choose a token for a search query for the exception (Chapter 4).

Listing 2.3: An Example Stack Trace (taken from [27])

```
Exception in thread "main" java.io.EOFException
at java.io.ObjectInputStream$PeekInputStream.readFully(ObjectInputStream.java: 2281)
at java.io.ObjectInputStream$BlockDataInputStream.readShort(ObjectInputStream.java:
    2750)
at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java: 780)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java: 280)
at HighScores.<init>(HighScores.java:45)
at HighScores.main(HighScores.java:151)
```

2.6 Graph Matching

In graph theory, *matching graphs* involves matching a set of independent edges along with their vertices [8]. Fig. 2.2, the adapted API usage graph of our toy example in Listing 2.2, 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* in the IDE, where we determine matching between two such corresponding usage graphs. We consider *maximum matching* in the graphs, and also estimate different *heuristic weights* (i.e., relative importance) for different types of matching (e.g., dependency, static relations) using a machine learning based approach (Section 6.3.4). For example, a data dependency matching is considered more important 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.

2.7 Logistic Regression

It is a probabilistic and statistical classification model that predicts binary or categorical outcomes based on a set of *predictor variables* (i.e., features). It is widely used in medical and social science fields. In our research, we use the regression model in association with a machine learning technique in order to estimate the relative weights (e.g., importance) of different metrics [61] proposed in each of the four studies. Logistic regression models the probabilities of different outcomes of a single trial as a function of predictor variables using a *logistic* function[4]. The logistic function is a common sigmoid function, $F(t)$, as follows:

$$F(t) = \frac{e^t}{e^t + 1}, \quad t = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \tag{2.3}$$

where $F(t)$ is a logistic function of a variable t, which is again a function of the predictor variables x_1 and x_2. Here, β_1, β_2 are coefficients, and β_0 is the intercept in the regression equation. The function always returns a value between zero and one, and thus provides a probabilistic measure for each type of outcomes for the trial. In our studies, we consider the coefficients (or their logarithmic transformations) of the regression equation generated during machine learning as the relative weights (i.e., relative importance) for the corresponding features (i.e., proposed metrics), and use them in our different ranking algorithms.

2.8 Page Rank Algorithm

PageRank algorithm by Lawrence Page and Sergey Brin is an efficient tool for ranking a list of web pages which are inter-linked with one another [19]. It is also widely used in other fields of research such as web spam detection, text mining, text summarization, word sense disambiguation, natural language processing and so on. The algorithm treats a hyper link in a web page to another website as a vote cast for that site, and it analyzes both incoming links and outgoing links of the page for the ranking. If the web page is highly hyper-linked (i.e., recommended) to other popular pages, it is also considered as a popular page, and vice versa. Thus, the *PageRank* score of the web page can be calculated as follows:

$$PR(A) = (1 - d) + d(\frac{PR(T_1)}{C(T_1)} + ... + \frac{PR(T_n)}{C(T_n)}) \tag{2.4}$$

Here, $PR(A)$ represents the *PageRank* of page A, $PR(T_i)$ represents the *PageRank* of pages T_i which link to page A, d refers to damping factor[5] which has a value between zero and one, and $C(T_i)$ is the number of outbound links on page T_i. In this research, we adapt this algorithm for the token network (e.g., Fig. 2.3)

[4] http://en.wikipedia.org/wiki/Logistic_regression
[5] The probability of jumping from one page to another by a visitor

developed from the stack trace of an exception, and determine the applicability of a token in a candidate search query for the exception (Chapter 4).

2.9 Summary

In this chapter, we introduced different terminologies and background concepts that would help one to follow the remaining of the thesis. We defined cosine similarity, a widely used text similarity matching technique throughout the thesis, degree of interest, a heuristic measure associated with stack trace, and DOM, a cross-platform document representation technique for an HTML or an XML document. We also discussed static relationships and data dependency graphs, stack trace token graph and graph-matching technique. Finally, we discussed logistic regression and PageRank algorithm that are used for metric weight estimation and determination of trace token rank respectively.

Chapter 3

An IDE-Based Context-Aware Meta Search Engine

Traditional web search does not consider context (i.e., surroundings, circumstances) of a programming problem (e.g., an error or an exception), and involves software developers into a *trial and error* based search activity for solution. It also forces them to switch frequently between their working environment (i.e., IDE) and the web browser, which is both distracting and time-consuming. In this chapter, we discuss our first study that proposes a solution to such problems.

The rest of the chapter is organized as follows. Section 3.2.1 discusses our proposed system model for IDE-based web search, and Section 3.2.2 presents our proposed content-based and context-based metrics and algorithms. Section 3.3 discusses experimental design, results and validation details, Section 3.4 identifies the possible threats to validity, Section 3.5 discusses the existing studies related to our research and finally, Section 3.6 summarizes the chapter with future works.

3.1 Introduction

Existing related studies focus on integrating commercial-off-the-shelf (COTS) tools into Eclipse IDE [71], recommending StackOverflow posts and then displaying them within the IDE [43, 70], embedding traditional web browser inside the IDE [36] and so on. Cordeiro et al. [43] propose an IDE-based recommendation system that recommends relevant StackOverflow posts for programming errors and exceptions. They extract a number of question and answer posts from StackOverflow data dump, and suggest those question posts that contain stack traces similar to that of an encountered exception in the IDE. Ponzanelli et al. [70] propose *Seahawk*, an Eclipse plugin, that analyzes the *context* (i.e., code under development) of the programming task at hand and recommends relevant StackOverflow posts in the IDE. It also visualizes different components (e.g., code segment, stack trace) of a recommended post through an embedded and customized web browser for legibility. Although these approaches have their inherent strengths, they also suffer from several limitations. First, they consider only one source– StackOverflow Q & A site for information, and thus the search scope is limited. Second, the developed corpus cannot be easily updated and is subjected to the availability of the data dump. For example, they use the StackOverflow data dump of September 2011, that means, it does not contain the posts created after September 2011. Thus their approach cannot suggest the StackOverflow posts discussing the recently introduced software bugs or errors (e.g., exceptions from new API libraries). Third,

they only consider either stack trace or source code under development as the *context* of a programming problem, which is partial and often does not help much. For example, the approach by Cordeiro et al. does not consider the code segment that triggers an exception and thus recommends solutions which might be non-applicable or even irrelevant to the code of interest given that the same exception could be triggered from different code context. Similarly, *Seahawk* [70] cannot recommend properly for the programming tasks associated with errors and exceptions as it does not analyze the stack traces reported by the IDE.

In this study, we propose a context-aware meta search solution, *SurfClipse*, to the encountered programming errors and exceptions in the IDE, which also addresses the concerns identified with the existing approaches. We package the solution as an Eclipse plugin prototype [25] which collects search results from a remotely hosted web service [30] and displays them within the IDE. The proposed approach (1) exploits the search and ranking algorithms of three reliable web search engines– Google, Bing and Yahoo and one programming Q & A site– StackOverflow through the use of their API endpoints, (2) provides both a content (e.g., error or exception message) relevance and context (e.g., stack trace, associated context code) relevance based ranking of the extracted results in step one, (3) collects the most recent posts and accesses the complete and extensible question and answer set of StackOverflow, and pulls solutions from a number of forums, discussion boards, blogs, programming Q & A sites and so on, and (4) demonstrates a potential use of web service technology for problem context-aware web search which can be easily leveraged by any IDE of any development framework.

We conduct experiments on *SurfClipse* using 75 programming errors and exceptions related to Eclipse plugin framework and standard Java applications, and compare with two existing approaches [43, 70] and three traditional web search engines. The proposed approach recommends correct solutions for 68 (90.66%) exceptions, which essentially outperforms the existing techniques and the keyword-based traditional search engines in terms of *recall* and other performance measures. We note from the experiments that neither stack trace nor context code alone can capture the complete context of an encountered exception, rather their combination represents a more precise context which is likely to help collect more relevant results. This work is a significantly extended and refined version of our earlier work [73], where we just outlined the idea of an IDE-based meta search engine with limited experiments and validations.

3.2 Proposed Approach for IDE-Based Context-Aware Meta Search

3.2.1 Proposed System Model

Fig. 3.1 shows the schematic diagram of our proposed approach for IDE-based context-aware web search. We implement our approach as an IDE-based search provider, and this section discusses the architectural design of our proposed system which includes the working modules, working modes and so on.

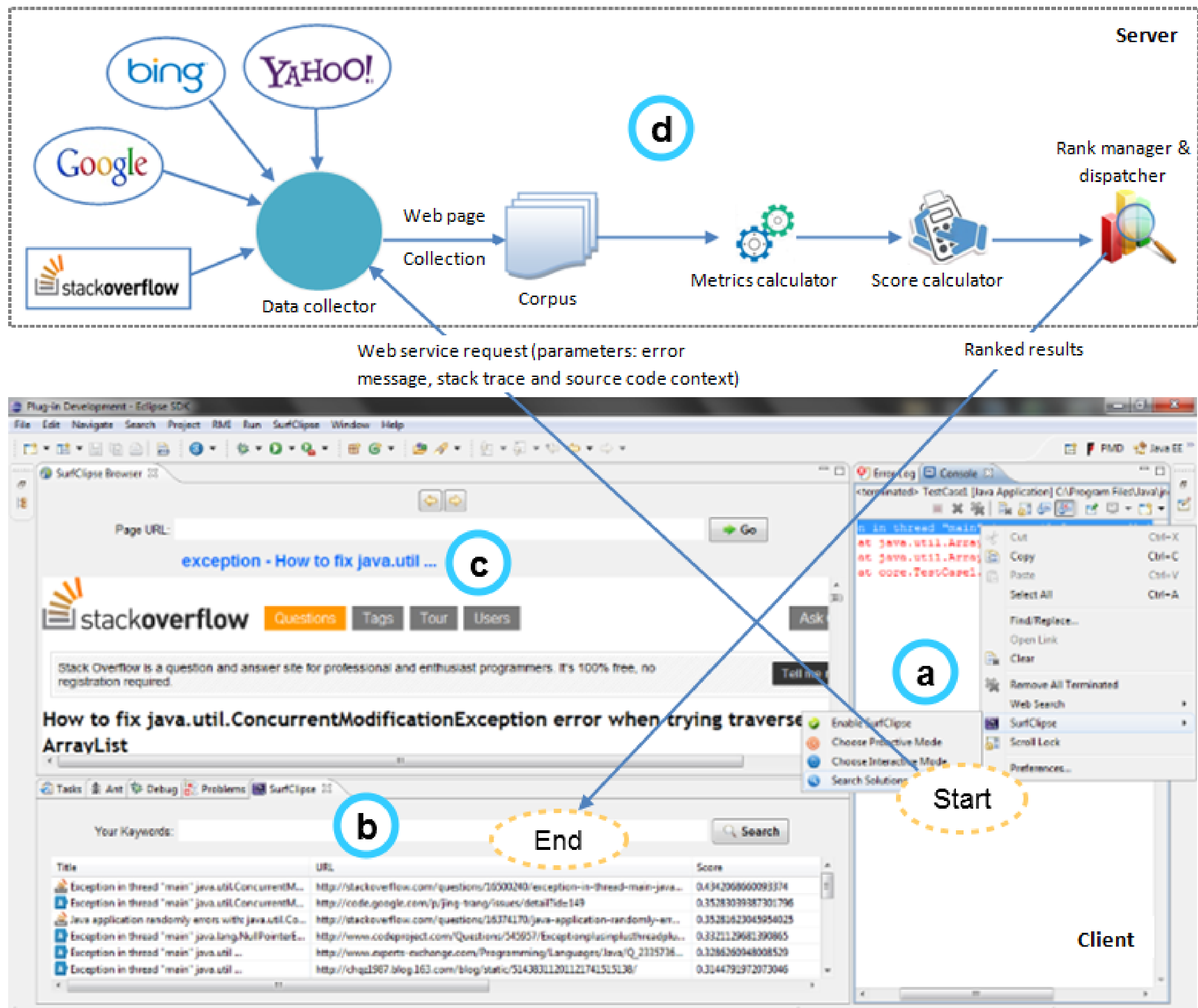

Figure 3.1: Schematic Diagram of Proposed Approach (SurfClipse)

Web Service Provider and Client Plugin

Our proposed system is based on client-server architecture and it has two major entities– Eclipse plugin (client) and web service provider (server). They communicate with each other through HTTP (Hyper Text Transfer Protocol) and facilitate web search within the IDE environment. Once a developer selects an encountered exception from *Console View* (e.g., Fig. 3.1-(a)) or *Error Log* in the IDE, the client plugin collects associated context– *stack trace* and *context code* (i.e., a segment in the code that triggers the exception), and generates a web search request to the service provider [30]. The service provider works as a meta search engine, that means, it collects results from multiple search engines against a search query and analyzes them to provide an enriched set of results. In our proposed model, *Data collector* module of the service provider (e.g., Fig. 3.1-(d)) collects results from three state-of-the-art search engines (Google, Bing and Yahoo) and one programming Q & A site (StackOverflow), and then accumulates the results to form a *Corpus*. The corpus is developed using about 100-150 results from different sources, and *Metrics calculator* module computes

different proposed metrics (Section 3.2.2) for each of the results in the corpus. The metrics capture the relevance of each result to the encountered exception as well as its context (i.e., stack trace, context code). Once metrics are computed, *Score calculator* calculates the final score of each result, and *Rank manager and dispatcher* ranks the results, and returns them to the client. The client plugin then captures them and displays within the IDE in a convenient way. It also facilitates the browsing of a result page through a customized web browser widget.

Plugin Working Modes

Eclipse plugin in the proposed model works in two modes– *interactive* and *proactive*. In interactive mode, a developer can select a search query by choosing a suitable phrase from the stack trace (e.g., Fig. 3.1-(a)) or associated context code of an encountered exception, and can make a web search request to the server (e.g., Fig. 3.1-(d)). The plugin also provides a flexible interface (e.g., Fig. 3.1-(b)) for keyword-based web search, where the developer can search with customized queries about the exception. In case of proactive mode, the web search request is initiated by the client plugin. In this mode, the plugin assigns a listener to the *Console View* which constantly checks for exception. Once an exception detected, the listener sends error message and context information to the plugin, and then plugin makes web search request to the service provider. Thus in this mode, the developer gets rid of the burden of carefully choosing the search query and making the search request manually, and she can concentrate on her current task without interruption.

Corpus Development

Reusing existing data and services in order to provide an enriched output is an interesting idea, and we use it for corpus development in our research. We exploit the available API services provided by three popular search engines– Google, Bing and Yahoo and one large programming Q & A site– StackOverflow to collect the top 30-50 ranked results from each of them against an encountered error or exception, and then use them to develop a corpus dynamically. The idea is– leveraging the existing search services and their recommendations to reduce search scope and to produce an effective solution set. Unlike a traditional search engine, which develops an index of all the result pages with some sort of relevance score against a query term, we store necessarily the complete HTML source of each result page. The source is parsed and analyzed for relevant stack traces, source code segments, and exception messages in the later phases for metric calculation.

3.2.2 Proposed Metrics

Search Engine Confidence Score (S_{sec})

According to *Alexa*[1], one of the widely recognized web traffic data providers, Google ranks first, Yahoo ranks fourth, Bing ranks nineteenth and StackOverflow ranks 67th based on the volume of their site traffic among

[1] http://www.alexa.com/topsites, Visited on September, 2013

all websites in the web in the year 2013. While these ranks indicate their popularity (i.e., site traffic) and reliability (i.e., users' trust) as information service providers, and existing studies show that different search engines perform differently, and even the same search engine performs differently based on the type of search query [59, 81], it is essentially reasonable to think that search results from different search engines with different ranks have different levels of acceptance. To determine the acceptance level of each search service provider, we conduct an experiment with 139 programming errors and exceptions[2]. We collect the top 10 search results against each of the exceptions from each search tool, and get their *Alexa ranks* [1]. We then consider the *Alexa ranks* of all result links provided by each search tool and calculate the average rank for a result link provided by them. The average rank for each search tool is then normalized and inversed which provides a value between zero and one, and we consider this value as a heuristic measure of confidence for the search tool. We use Equations (3.1) and (3.2) to get the *search engine confidence* for a result link.

$$R_{i,normal} = \frac{\bar{R}_i}{\sum_{i=1}^{n} \bar{R}_i} \tag{3.1}$$

$$S_{i,sec} = \frac{\frac{1}{R_{i,normal}}}{\sum_{i=1}^{n} \frac{1}{R_{i,normal}}} \tag{3.2}$$

Here, $\bar{R}_i$ represents the average *Alexa rank* for each search tool results, $R_{i,normal}$ is the normalized version of $\bar{R}_i$ and $S_{i,sec}$ refers to the final confidence score for each search tool based on *Alexa search traffic* statistics. We get a normalized confidence of 0.29 for Google, 0.35 for Bing, 0.36 for Yahoo and 1.00 for StackOverflow. Given that StackOverflow is a popular programming Q & A site that has drawn the attention of a vast programming community (1.9 million[3]) and contains about 12 million questions and answers, its result pages have the maximum confidence. The idea is– the occurrence of a result link in multiple search engines against a single query would issue the corresponding confidence scores of the search engines to the link. Thus if a result occurs in all search provider results, it gets a confidence score of 2.00; however, the confidence scores of all results in the corpus are normalized for practical use.

Content Matching Score (S_{cms})

During errors or exceptions, an IDE or Java framework generally issues notifications from a fixed set of error or exception messages unless a developer handles the exception manually. Thus there is a great chance that a web page titled with an error or exception message similar to the search query (i.e., error message of the encountered exception) would discuss a programming problem similar to the one encountered by the developer, and would contain relevant and useful information. We propose a metric, *Title to Title Similarity* (S_{tts}), that measures the content similarity between the query message and the title of each result page in the corpus. We use *cosine similarity measure* for this purpose which returns a value between zero (i.e., completely dissimilar) and one (i.e., exactly similar). As we noted that the result title may not always

[2] http://homepage.usask.ca/∼mor543/query.txt
[3] http://en.wikipedia.org/wiki/Stackoverflow, Visited on September, 2013

provide enough information about the discussed problem(s) in the page, the body content of the page needs to be consulted. We consider stack traces, source code segments and discussion texts extracted from the page as legitimate sources of information about the discussed exceptions, and propose two cosine similarity-based metrics– *Title to Context Similarity* (S_{tcx}) and *Title to Description Similarity* (S_{tds}). *Title to Context Similarity* score determines the content relevance between the query error message and the extracted context (e.g., stack traces, associated code snippets) of the discussed exceptions, and *Title to Description Similarity* score denotes the possibility of the occurrence of query error message in the discussion texts.

According to Arif et al. [31], terms contained in different parts of a document deserve different levels of attention. For example, a phrase in the page title is more important than a phrase in discussion texts for specifying the subject matter of the document. We reflect this idea in content matching, and assign different weights to different content similarity scores. We use Equation (3.3) to determine the content relevance between the search query and each result page in the corpus.

$$S_{cms} = \alpha \times S_{tts} + \beta \times S_{tcx} + \gamma \times S_{tds} \tag{3.3}$$

Here, α, β and γ are the assigned weights to result page title, extracted context (stack trace and code snippet) and discussion texts respectively, and they sum to one. Given that the similarity scores are generated from cosine-based measures, *Content Matching Score* always ranges from zero (i.e., completely irrelevant) to one (i.e., completely relevant).

Stack Trace Matching Score (S_{stm})

In the resolution of programming errors or exceptions, the associated context such as stack trace reported by the IDE plays an important role. A stack trace contains the encountered error or exception type, a system message and a list of method call references. In this research, we consider an incentive to the result pages in the corpus containing stack traces similar to that of the encountered (i.e., target) error or exception. We consider both the lexical and structural perspectives of a stack trace, and propose two metrics– *Lexical Similarity Score* (S_{lex}) and *Structural Similarity Score* (S_{stc}), in order to determine the relevance between stack traces. The information in the stack trace can be categorized into two– a detailed technical error message (first part) and possible locations of the exception (second part). We parse the exception name and the error message from the first part and extract package name, class name and method name tokens from each of the call references of second part, and develop a *token set* for the stack trace. We use this token set to determine the lexical relevance between the stack trace of the target exception and a candidate stack trace from a result page, and we use *Cosine Similarity Score* for the purpose. It should be mentioned that we do not decompose the camel case tokens into granular levels (i.e., granularization introduces false positives) in order to perform meaningful similarity checking, which makes the relevance checking effective and useful.

Method call references and their sequence in the stack trace provide important clues about the location of the target exception and thus they also can be leveraged for determining relevance between two stack traces. We calculate *Degree of Interest Score* for each reference in the target (i.e., encountered) stack trace

using Equation (2.2) (Section 2.2) and use them to determine *structural relevance* with the candidate stack traces from the result pages. The idea is to determine the occurrence of method call references of the target stack trace in the candidate stack traces. However, complete matching between two references may not be likely and we exploit the idea of *confidence coefficient* proposed by Cordeiro et al. [43]. We get the *Structural Similarity Score* between the stack trace of the encountered exception and a candidate stack trace using Equations (3.4) (proposed by Cordeiro et al. [43]) and Equation (3.5).

$$ms_i = S_{doi} \times c_i \tag{3.4}$$

$$S_{stc} = \frac{1}{N}\sum_{i=0}^{N} ms_i \tag{3.5}$$

Here, ms_i denotes the matching score between two references, c_i refers to confidence coefficient and N represents the number of method call references in the stack trace of the encountered exception. We consider a heuristic value of 0.50 for c_i if *method name* and *class name* tokens match between two call references, and consider 0.90 when *package name* tokens also match. We consider c_i=1.00 only when all four tokens–*package name, class name, method name*, and *line number* match between two call references. The heuristic values are inspired by those of Cordeiro et al. [43]. Once both lexical relevance and structural relevance are found, we get the *Stack Trace Matching Score* using the following equation:

$$S_{st} = \delta \times S_{stc} + \sigma \times S_{lex} \tag{3.6}$$

Here, δ and σ denote two heuristic weights assigned to *structural similarity* and *lexical similarity* scores respectively, and they sum to one. *Stack Trace Matching Score* values from zero and one, where zero represents total irrelevance and one represents the complete relevance between two stack traces.

Source Code Context Matching Score (S_{ccx})

Sometimes, stack trace alone may not provide enough information about the encountered exception for analysis, and the source code triggering the exception needs to be consulted. In programming Q & A sites, forums and discussion boards, users often post source code snippets related to the exception besides the stack traces for clarification. We are interested to check if the code snippets in the result page are similar to the source code associated with the encountered (i.e., target) exception in the IDE. This coincidence is possible with the notion that developers often reuse code snippets from different programming Q & A sites, forums or discussion boards in their programs directly or with minor modifications. Therefore, a result page containing code snippets similar to the code associated with the target exception is likely to discuss relevant issues that a developer needs to know in order to solve the target exception. We consider the code surrounding the exception location in the source file in the IDE as the *source code context* of the target exception, and use a *code clone detection technique* [75] to determine its relevance with the code snippets extracted from the result pages. The idea is to identify the *longest common subsequence of tokens* between two token sets extracted

from two different code snippets. We use Equation (3.7) in order to determine the relevance between the *context code* (i.e., source code context) of the target exception and a code snippet from the result page.

$$S_{ccx} = \frac{|S_{lcs}|}{|S_{total}|} \tag{3.7}$$

Here, S_{lcs} denotes the longest common subsequence of tokens, and S_{total} denotes the set of tokens extracted from the code block considered as the context of the encountered exception in the IDE. The *Source Code Context Matching Score* values from zero to one.

StackOverflow Vote Score (S_{so})

StackOverflow, a popular programming Q & A site with 1.9 million users, maintains a score for each question, answer and comment posted by the users, and the score can be considered as a social and technical recognition of their merit [65]. In StackOverflow, a user can up-vote any question or answer post if she likes something about them, and can also down-vote if the post content seems erroneous, confusing or not helpful. Thus the difference between up-votes and down-votes from a vast community of technical users, the score of post, is considered as an important metric for evaluation of the quality of the solution posted. In our research, we consider such scores of the posted question and answers in the result page from StackOverflow, and calculate *StackOverflow Vote Count* using Equation (3.8). We then normalize the vote count and get *StackOverflow Vote Score* using Equation (3.9).

$$SO_k = \sum_{\forall p \epsilon P} V_p \tag{3.8}$$

$$S_{so} = \frac{SO_k - \lambda}{max(SO_k) - \lambda} \tag{3.9}$$

Here, SO_k refers to the StackOverflow vote count for a result page, V_p denotes the vote count for a post in the page, p refers to any post, and P denotes the set of question and answer posts found in a result page. λ denotes the minimum vote count, $max(SO_k)$ represents the maximum vote count and S_{so} is the normalized *StackOverflow Vote Score* for the result link. The score values from zero (i.e., least significant) to one (i.e., most significant) and it indicates the relative quality or popularity of the StackOverflow link in the eyes of a large crowd of technical users.

Search Traffic Rank Score (S_{str})

The amount of search traffic to a site can be considered as an important indicator of its popularity. In this research, we consider the relative popularity of the result links extracted from different search engines. We use the statistical data from two popular site traffic control organizations– *Alexa* and *compete* through their provided APIs and get the average popularity rank for each result link. Then, based on these ranks, we provide a normalized *Search Traffic Rank Score* to each result link between zero and one considering minimum and maximum ranks found.

3.2.3 Result Scores Calculation

The proposed metrics (Section 3.2.2) focus on four aspects of each result– content relevance, context relevance, popularity and search engine confidence, and we consider those aspects for the calculation of final scores. We use *Content Matching Score* of each result page as its *Content Relevance,* (R_{cnt}) with the encountered exception. In this research, we consider stack trace and code segment triggering the exception as the context of the encountered exception in the IDE, and use *Stack Trace Matching Score* and *Source Code Context Matching Score* to determine the *Context Relevance, (R_{cxt})* of each result page. Both stack trace and context code carry different levels of significance and we assign two heuristic weights to the matching scores in order to get the context relevance score.

$$R_{cxt} = w_{st} \times S_{stm} + w_{cc} \times S_{ccx} \tag{3.10}$$

Here, w_{st} and w_{cc} are the assigned weights to S_{stm} and S_{ccx} respectively, and they sum to one which gives R_{cxt} normalized, and it values from zero to one.

StackOverflow Vote Score and *Search Traffic Rank Score* are considered as the estimates of popularity of each result link from different viewpoints, and they deserve different levels of attention. In the calculation of *Popularity Score*(S_{pop}) of each result link, we assign two different heuristic weights to these metrics.

$$S_{pop} = w_{so} \times S_{so} + w_{sr} \times S_{str} \tag{3.11}$$

Here, w_{so} and w_{sr} represent the assigned weights to S_{so} and S_{str} respectively, and they sum to one; this gives S_{pop} a normalized value from zero (i.e., the least popular) to one (i.e., the most popular).

Confidence of each result, obtained from the associated search engines, can be considered as a support measure for the result link against a search query. We consider *Search Engine Confidence Score* as the confidence of each result. Thus the four component scores associated with four aspects can be combined using Equation (3.12) in order to get the final score for each result.

$$S_{final} = \sum_{\exists w \epsilon W, \exists RS \epsilon (R_{cnt}, R_{cxt}, S_{pop}, S_{sec})} w \times RS \tag{3.12}$$

Here, RS denotes a measure of content relevance, context relevance, popularity or confidence of a result in the corpus, and w denotes the individual weight (i.e., importance) associated with each RS. We assign a heuristic weight of 0.35 to content-relevance, 0.85 to context-relevance, 0.20 to popularity and 0.10 to the impression of the result link with the search engines. We choose these heuristic weights based on our extensive and iterative controlled experiments with a subset of all exceptions, manual analysis on the experimental results, discussion among the authors, and also some helpful ideas from the existing study [43]. While these heuristic values might seem a bit arbitrary, we find the combination to be the best in our experiments to represent the relative importance of different aspects of the final score for a result. Once the final scores are found, the results are sorted in a descending order and the top twenty or thirty of them are returned to the requesting client.

3.3 Experimental Design, Results and Validation

3.3.1 Dataset Preparation

We collect the *workspace logs* of Eclipse IDE from six graduate research students of Software Research Lab, University of Saskatchewan, and extract the exceptions (e.g., error message, stack trace) occurred during the last six months. We collect a list of 214 stack traces from them. We find that most of the stack traces are duplicate of one another, and we choose 38 distinct stack traces involving 44 *Exception classes*, which are mostly related to Eclipse plugin development framework. To include exceptions related to standard Java application development and prepare a balanced dataset, we choose 37 exceptions from a list of common Java exceptions [24]. We then generate some of those exceptions using code examples, and also perform exhaustive web search to collect stack traces and source code context associated with those exceptions. We finally get a list of 75 exceptions [24] associated with 75 stack traces [24] and 37 contextual code blocks [24], which we use as the dataset for different experiments. It should be noted that we cannot collect helpful context code for the exceptions extracted from the *workspace logs* of the IDE. We collect the most appropriate solutions for the exceptions with the help of different available search engines such as Google, Bing, Yahoo and Ask. Given that checking relevance of a solution is controlled by different subjective factors, we select the solution list carefully. First, one of the authors performs exhaustive web searches for two days and collects a potential list of solutions for the exceptions which are shared with other authors. The other authors review the exception information (e.g., stack trace, code context) and the solution list independently, and provide their feedback about the selection of solution list. Then, the suggestions of all authors are accumulated to finalize a solution set [24] for the exceptions in the dataset.

3.3.2 Investigation with Eclipse IDE

We analyze the features provided by Eclipse IDE in order to find out how often developers can get necessary help in solving the encountered errors and exceptions. It is interesting to note that the IDE provides nice debugging tools for them to analyze the exceptions through check-pointing, but they do not help much in platform-level exceptions associated with different runtime libraries and configuration files. For example, if a Java application tries to consume more memory space than alloted, Java framework would issue this exception message– *java.lang.OutOfMemoryError: Java heap space*, and the debugging tools have a very little opportunity to help with this problem. There is also an internal web browser widget in recent versions of Eclipse IDE; however, it is intended for browsing web pages and is defaulted to *Bing Search* which does not consider the context of the encountered exceptions, and thus cannot help the developers much effectively.

3.3.3 Search Query Formulation

Every web search request from Eclipse client plugin has three components– query for search engines, stack trace and context code. Our proposed solution works in two modes– interactive and proactive. In the case of interactive mode, the developer forms the search query by carefully selecting keywords from the exception message and context information of the exception, whereas the plugin is responsible for generating the search query itself in case of proactive mode. This section discusses the query formulation technique used by our tool during its proactive mode.

In this research, we consider both the stack trace and the code segment likely responsible for the encountered exception as the context of the exception. We thus collect information from the context besides the exception message in order to develop a search query for the exception. Traditional search engines generally do not allow[4] or perform poorly with long queries [59, 81], and in order to collect results from them, we use a sophisticated technique to describe the exception and its context in terms of few tokens. We capture the exception message containing exception class name and collect five class name and method name tokens with the highest *Degree of Interests* from the stack trace [43]. We also extract five most frequent method calls and imported class names from the context code using an *ASTParser library*[5] (i.e., in case of compilable code) and a custom island parser (i.e., in case of uncompilable code) [70]. We then combine the extracted method and class name tokens from both context to develop a unique list of query terms and add the list to the exception message. We note that the exception message itself returned by the IDE is a good descriptor of the exception; however, we filter the message and discard different irrelevant components such as absolute file path, URL and so on. Thus the filtered exception message and the list describing the context of the exception develop the search query, and the client plugin use it to collect results (i.e., for corpus development) from different search engines in proactive mode.

3.3.4 Performance Metrics

Given that our proposed approach is aligned with the research areas of information retrieval and recommendation systems, we use a list of performance metrics from those areas as follows.

Mean Precision (MP)

While *precision* denotes the fraction of retrieved results that are relevant to the query, *Mean Precision* is the average of that measure for all queries in the dataset.

[4] http://en.wikipedia.org/wiki/Web_search_query
[5] http://code.google.com/p/javaparser/

Table 3.1: Results of Experiments on Two Working Modes of Proposed Approach

Mode	Metric	Top 10[1]	Top 20[2]	Top 30[3]
Interactive	Mean Precision (MP)	0.1229	0.0736	0.0538
	MFFP	1.2400	1.2400	1.2400
	MRR	0.4604	0.4648	0.4669
	TEF[4]	59(75)	64(75)	**68**(75)
	Recall (R)[5]	78.66%	**85.33%**	**90.66%**
Proactive	Mean Precision (MP)	0.0866	0.0529	0.0380
	MFFP	1.2400	1.2400	1.2400
	MRR	0.4009	0.4048	0.4054
	TEF	51(75)	55(75)	56(75)
	Recall (R)	68.00%	73.33%	**74.66%**

[1]Metrics for the top 10 results, [2]Metrics for the top 20 results

[3]Metrics for the top 30 results, [4]Number of exceptions fixed

[5]Percentage of the exceptions fixed

Mean First False-Positive Position (MFFP)

First False-positive Position (FFP) is the rank of first false-positive result in the ranked list. *Mean First False-Positive Position* measures the average first false-positive position for each query in the query set.

Mean Reciprocal Rank (MRR)

Reciprocal Rank is the multiplicative inverse of the rank of first relevant result. *Mean Reciprocal Rank* is a statistical measure that averages the *Reciprocal Rank* for each query in the query set.

Recall (R)

Recall denotes the fraction of the relevant results that are retrieved. In our experiments, we consider *recall* as the percentage of the test cases (i.e., exceptions) for which the solutions are recommended correctly.

3.3.5 Experimental Results on Proposed Approach

In our experiments, we conduct search with each exception in the dataset and collect the top 30 results. We consider both working modes– interactive and proactive, and analyze the results using different performance metrics (Section 3.3.4). Tables 3.1, 3.2 and 3.3 show the results of the experiments conducted on our approach.

Table 3.1 shows a comparative analysis between *interactive* mode and *proactive* mode of search of the proposed approach. Here, we see that our tool performs relatively better in interactive mode than proactive mode in terms of different performance metrics such as Mean Precision (MP), Mean First False-Positive Position (MFFP), Mean Reciprocal Rank (MRR) and Recall (R). We also note that *proactive version* can recommend correct solutions for 56 exceptions in total whereas *interactive version* can recommend for 68 out

of 75 exceptions, which gives a recommendation accuracy of 90.66% for our approach. Given that formulating search query is one of the decisive factors for the performance of *interactive approach*, we manually select a list of search-friendly keywords from the context of each exception as the search query. The query collects a richer set of initial results (i.e., for corpus development) from multiple search engines than that of *proactive approach*, where the search keywords are not fine-tuned for search engines.

Table 3.2 investigates the impacts of different aspects– content relevance, context relevance, popularity and search engine confidence of the result link in the ranking of search results. It shows how the incremental association of different aspects can improve the search results in terms of performance metrics such as Mean Precision (MP), Recall (R) and so on. We note that the proposed approach provides the highest MP and the highest recall when all four aspects are considered during score calculation of a result rather than a single aspect such as *content-relevance*. For example, it performs the best (e.g., 90.66%) in terms of *accuracy* (e.g., TEF (No. of total exceptions fixed), R (% of exceptions fixed)) when all four dimensions of the result score are considered, which shows the potential of exploiting associated context information besides search query (i.e., error or exception message) during search.

Table 3.3 compares the experimental results achieved against two different sets of exceptions– one with exception messages and stack traces (i.e., Set A) and the other with exception messages, stack traces and context code (i.e., Set B). Here, we see that Set B, that considers context code besides stack trace and error message of an exception, achieves higher accuracies (e.g., 97.29% and 86.48%) than Set A (e.g., 84.21% and 63.16%) in both working modes. It also gets better results in terms of other performance metrics such as Mean Precision (MP) and Mean Reciprocal Rank (MRR). The findings show that a combination of context code and stack trace can better specify the context of the exception rather than stack trace only, and thus, by exploiting the context, the proposed approach can recommend more solutions for the set that captures the combination than the one that do not capture.

3.3.6 Comparison with Existing Approaches

We compare the results of our proposed approach against two existing IDE-based recommendation systems– context-based recommendation system by Cordeiro et al. [43] and *Seahawk* by Ponzanelli et al. [70]. Both of them collect data from StackOverflow data dump and recommend StackOverflow posts taking the current context of the search into consideration. They select suitable tokens from either stack trace or context code to describe the problem context in the search query, and recommend solutions in a proactive fashion. We implement both of the existing methods and use them for experiments.

To implement the approach proposed by Cordeiro et al. [43], we use the exception name from each exception test case and collect 100 top voted StackOverflow posts discussing about that exception, and develop a corpus for the test case. We download the page source of each item in the corpus and create an *Apache Lucene Index* for the corpus. We then use the index and *Lucene search engine* to retrieve the relevant posts against the search query associated with the test case. From *Lucene*, we also collect the *retrieval score*

Table 3.2: Experimental Results for Different Ranking Aspects

Mode	Score Combination	Metric	Top 10	Top 20	Top 30
Interactive	Content (R_{cnt})	MP	0.0899	0.0607	0.0481
		TEF	43	55	65
		Recall (R)	57.33%	73.33%	86.66%
	Content (R_{cnt}) and Context (R_{cxt})	MP	0.11428	0.0699	0.0514
		TEF	58	63	66
		Recall (R)	77.33%	84.00%	**88.00%**
	Content (R_{cnt}), Context (R_{cxt}), and Link Popularity (S_{pop})	MP	0.1157	0.0699	0.0519
		TEF	57	63	66
		Recall (R)	76.00%	84.00%	**88.00%**
	Content (R_{cnt}), Context (R_{cxt}), Link Popularity (S_{pop}) and Result confidence (S_{sec})	MP	**0.1229**	0.0736	0.0538
		TEF	59	64	**68**
		Recall (R)	78.66%	**85.33%**	**90.66%**
Proactive	Content (R_{cnt})	MP	0.0871	0.0528	0.0371
		TEF	46	54	56
		Recall (R)	61.33%	72.00%	74.66%
	Content (R_{cnt}) and Context (R_{cxt})	MP	0.0785	0.0499	0.0376
		TEF	45	52	55
		Recall (R)	60.00%	69.33%	73.33%
	Content (R_{cnt}), Context (R_{cxt}), and Link Popularity (S_{pop})	MP	0.0857	0.0542	0.0381
		TEF	49	55	56
		Recall (R)	65.33%	73.33%	74.66%
	Content (R_{cnt}), Context (R_{cxt}), Link Popularity (S_{pop}) and Result confidence (S_{sec})	MP	0.0886	0.0529	0.0380
		TEF	51	55	56
		Recall (R)	68.00%	73.33%	**74.66%**

based on *Vector Space Model* for each retrieved post. We calculate the *structural score* and *lexical score* of each post considering their stack traces and then normalize them. Finally, we add all three scores for each post to get the final score.

In case of *Seahawk* proposed by Ponzanelli et al. [70], we collect the ten most frequent tokens (e.g., method name, class name) from context code of an exception test case as the search query, and retrieve relevant posts from StackOverflow containing suitable code examples or discussions relevant to the corresponding exception. Given that *Apache Solr* is a search service provider using *Apache Lucene* as the core search engine, we use *Apache Lucene* to collect relevant results against a search query. Basically, we reuse the previously developed indexes of StackOverflow posts and collect the relevant posts as well as their relevance scores.

Table 3.4 (top part) shows a comparative analysis between the results of two existing approaches– Cordeiro et al. [43] and Ponzanelli et al. [70], and our proposed approach. The working principles of the existing approaches are similar to that of *proactive* version of our tool, and thus we compare them to the *proactive* version. Here, we can see that both of the existing approaches perform poorly in terms of all performance

Table 3.3: Results of Experiments on Multiple Sets

Mode	Metric	Set A[1](38)	Set B[2](37)
Interactive	Mean Precision (MP)	0.0404	0.0604
	MFFP	1.0000	1.4864
	MRR	0.2695	0.6697
	TEF	32	36
	Recall (R)	**84.21%**	**97.29%**
Proactive	Mean Precision (MP)	0.0263	0.0450
	MFFP	1.0000	1.4864
	MRR	0.2563	0.5585
	TEF	24	32
	Recall (R)	**63.16%**	**86.48%**

[1] Contains exception message and stack trace.

[2] Contains exception message, stack trace and code context.

metrics compared to our approach. In the best case, they can recommend solutions for 24.00% and 18.92% of the exceptions respectively. The findings show that depending on a single information source for exception is not a good choice, and the combination of stack trace and source code context is a preferable choice to either any of them alone for reflecting the context of an exception during search.

3.3.7 Comparison with Existing Search Engines

We compare the results of our proposed approach against four available search engines– Google, Bing, Yahoo and StackOverflow (i.e., provides an API for search within StackOverflow). The interactive mode of our approach allows the developer to provide a search query which resembles with working principles of the search engines. We develop search query for each exception using suitable tokens from the technical error message and the context, and use them to collect results from the search engines as well as from the proposed approach. It should be mentioned that the search query is simply used to develop the corpus in case of the proposed approach, and final ranking of the results are determined with the help of ranking algorithms using the automatically extracted context information from the IDE. We collect the top 30 results from each search provider and look for expected solutions identified previously (Section 3.3.1).

Table 3.4 (bottom part) shows the comparative analysis of results from different search engines and our proposed approach. Here, we see that existing search engines can recommend solutions for at most 58 (77.33%) out of 75 exceptions, where the proposed approach can recommend for 68 (90.66%) exceptions. We also note that *Google* performs slightly better than our approach in terms of Mean Precision (MP), but recommends correct solutions for only 57 exceptions.

Table 3.4: Comparison with Existing Approaches and Search Engines

Mode	Recommender	#TE[1]	Metric	Top 10	Top 20	Top 30
Proactive	Cordeiro et al. [43]	75	MP	0.0202	0.0128	0.0085
			TEF[2]	15	18	18
			R[3]	20.00%	24.00%	**24.00%**
	Proposed Approach	75	MP	0.0886	0.0529	0.0380
			TEF	51	55	**56**
			R	68.00%	73.33%	**74.66%**
	Ponzanelli et al. [70]	37	MP	0.0243	0.0135	0.0099
			TEF	7	7	7
			R	18.92%	18.92%	**18.92%**
	Proposed Approach	37	MP	0.1000	0.0621	0.0450
			TEF	30	**32**	**32**
			R	81.08%	86.48%	**86.48%**
Interactive	Google	75	MP	**0.1571**	0.0864	0.0580
			TEF	57	57	57
			R	76.00%	76.00%	**76.00%**
	Bing	75	MP	0.1013	.0533	0.0364
			TEF	55	58	58
			R	73.33%	77.33%	**77.33%**
	Yahoo	75	MP	0.0986	0.0539	0.0369
			TEF	54	57	57
			R	72.00%	76.00%	**76.00%**
	StackOverflow	75	MP	0.0226	0.0140	0.0097
			TEF	14	17	17
			R	18.66%	22.66%	**22.66%**
	Proposed Approach	75	MP	**0.1229**	0.0736	0.0538
			TEF	59	64	**68**
			R	**78.66%**	**85.33%**	**90.66%**

[1]Number of exceptions used for the experiment, [2]Number of total exceptions fixed.
[3]Percentage of the exceptions fixed

Given that *selection of appropriate query terms* is an essential precondition for successful search, we conduct another experiment with those search engines using two scenarios–keywords from only exception message, and keywords both from exception message and exception context. Table 3.5 shows the results of those two scenarios. Here, we see that the keyword-based query that considers the exception context, provides more relevant results than the one that does not consider. Thus the performance of the traditional search engines is subjected to the selection of search keywords, and the appropriateness of this selection entirely depends on the developer's skill. In our experiments, we choose the search keywords carefully

Table 3.5: Impact of Context, and Common and Unique Results from Search Engines

Search Query	Common	Google Only	Bing Only	Yahoo Only
Exception Message Only	32	09	16	18
Message and Context of Exception	47	09	11	10

which provides the better results (e.g., precision) for Google, but it cannot be taken for granted given the uncertainty in query selection. On the other hand, it is interesting to note that our approach, for the same set of queries, can recommend correct solutions for more exceptions with a little compromise in the precision, and the developers get rid of the burden of choosing appropriate tokens from the context. They can select an encountered exception for search from *Console View* in the IDE, and the plugin itself captures the detailed context of the exception (e.g., stack trace and context code) to recommend relevant results, whereas Google depends entirely on the developers for the context-based information.

Given the precise results from Google search engine, and the correlation between Mean Precision (MP) and Recommendation Accuracy (e.g., R) observed at Table 3.2, one may argue that only Google results should be considered for corpus development in the proposed approach. In our research, we investigate whether such corpus is likely to contain the correct solutions for more exceptions or not. We develop corpus for each of 75 exceptions collecting the top 100 results from Google search API against the selected exception, and apply the proposed ranking algorithms. From Table 3.4 (bottom part) we find that the Google corpus-based approach can recommend correctly for at most 57 exceptions. Moreover, Table 3.5 shows that each search provider contains some unique recommendations which cannot be exploited if we consider only one search engine. Therefore, the idea of accumulating search results from multiple search engines for corpus development is promising, and it ensures the maximum *Recall (R)* for our approach by leveraging the existing search services.

We also investigate into why the proposed approach provides slightly less precise results compared to Google. Given that our approach involves scraping of semi-structured data from the result web page, it may sometimes fail to extract the exception context information properly if the page does not contain the information in the expected tags (e.g., *code, pre, blockquote*). From our manual analysis with ten cases having the most precise and the least precise recommendations, we find that recommended pages containing context information (e.g., stack trace, context code snippets) relevant to the exception of interest (i.e., query exception) are likely to rank higher than those which do not contain such information. In case of the least precise results, the recommended pages contain that information either in an unstructured way which is difficult to extract or they do not contain it at all. Since our proposed approach emphasizes on the context of the problem discussed in a result page, it does not perform well for those cases. Therefore, improvement of the context information extraction techniques from web page can help to enhance the *precision* of the proposed approach, which we consider as a scope for future study.

3.4 Threats to Validity

During the research, we identify a few threats to validity which we discuss in this section. First, the proposed approach still does not provide the search results in real time. Given that the approach involves into scraping of web page content for context of the discussed problem, it takes 20-25 seconds in average to return the recommendations. We applied Java based multi-threading to speed up the computation; however, the approach can be made returning results in real time by more extensive parallelization on the web server, and we have already designed it for multi-core systems.

During the experiments, we note that the existing search engines evolve rapidly, especially Google, within days and weeks, and the recommendations from the search engine vary over time for the same query. Therefore, the statistics from the experiments with the search engines are very likely to change. Given that our approach exploits the *live API services* from them, it would also evolve, and it is also subjected to the strength and weaknesses of the search engines. However, adoption of meta search based approach is likely to aggregate the strength and mitigate the weaknesses of each individual search engine as we showed the effectiveness.

Most of the programming errors and exceptions we selected for the experiments are frequently encountered by the developers , and their solutions are also widely discussed in the web. One may argue if the wide availability of those solutions contributes to the better performance of the proposed approach or not. Our approach does not differentiate between frequent and rare programming exceptions, and it returns the relevant recommendations as long as sufficient data are collected from the search engines. However, the approach is subjected to the availability of the appropriate context information (e.g., stack traces, context code) in the web page for relevance checking.

3.5 Related Work

Existing studies related to our research focus on integrating commercial-off-the-shelf (COTS) tools into Eclipse IDE [71], recommending StackOverflow posts and displaying them within IDE environment [43, 70], recommending previously visited web pages [76] and open source codes [32], embedding traditional web browser inside the IDE [36] and so on. Poshyvanyk et al. [71] integrate Google Desktop Search API into Eclipse environment to facilitate customized search within the IDE, which can be leveraged by different software maintenance activities. Cordeiro et al. [43] propose an IDE based recommendation system for programming exceptions. They extract question and answer posts from StackOverflow data dump, and suggest posts proactively relevant to an encountered exception in the IDE by analyzing the stack trace reported by the IDE. In contrast to StackOverflow data dump, our research exploits the existing web search and StackOverflow API services to collect filtered and relevant data from multiple sources. It also considers context code of the exception besides the stack trace as well as popularity and search engine confidence of the result link for ranking. Ponzanelli et al. [70] propose another Eclipse IDE based recommendation system,

Seahawk, that analyzes the code under development in the IDE and recommends relevant StackOverflow posts containing code examples and discussions helpful to the coding. However, it does not consider the stack trace as a component of problem context, and thus its recommendations are not sufficient for programming tasks associated with errors and exceptions. Brandt et al. [36] embed a custom code search engine, *Blueprint*, in Eclipse IDE and conduct a user study in the laboratory environment to investigate whether IDE-based browser can help developer productivity compared to stand-alone web browser. They conclude that the tool helped the developers significantly to write better code and to find code examples, and task-specific search interface can greatly influence the web search usage. Our research is related to it in the sense that we also attempt to address the context-switching issues through IDE based web search features and suitable user interfaces. Sawadsky et al. [76] propose *Reverb*, a tool that considers the code under active development within the IDE, and proactively recommends previously visited and relevant web pages from the browsing history. Bajracharya et al. [32] propose *Sourcerer*, an open source code search engine that considers both *TF-IDF* and structural relationships among the code elements to recommend Java classes from 1500 open source projects. Both Sawadsky et al. and Bajracharya et al. exploit lexical and structural features of the source code for recommendation. In our research, we apply similar set of features of the code with the focus on matching local code context of an encountered exception in the IDE against that of the exceptions and programming problems discussed in the web pages for relevant recommendation within the IDE. For some of the existing approaches [43, 70], the detailed comparison results can be found in Section 3.3.6.

3.6 Summary

To summarize, we propose a novel IDE-based context-aware web search solution for programming errors and exceptions in the IDE. The approach exploits three reliable web search engines and a programming Q & A site through their API endpoints for search results, and it considers both the problem content and problem context during search. It also considers the popularity and the impression of each result to different search engines during ranking of the results. We conduct experiments on our approach with 75 programming errors and exceptions, and compare with two existing approaches, three search engines– Google, Bing and Yahoo and StackOverflow search feature. Experiments show that our approach outperforms the existing approaches, search engines and StackOverflow search feature in terms of *recall* and other performance metrics. Experiments also show that inclusion of all types of context information (i.e., detailed context) of an exception during search can improve the accuracy of a recommendation system. Given the decisive role of a search query in our proposed approach (i.e., especially in interactive mode), the search can be further complemented with automated support in query development. In Chapter 4, we thus propose an approach for context-aware search query recommendation. In order to validate the applicability of our proposed meta search based approach in real life problem solving, an extensive user study should be conducted which is also later done in Chapter 7.

Chapter 4

Context-Aware Search Query Recommendation

The first study in Chapter 3 explores the potential for the use of technical details (e.g., stack trace) and programming context (e.g., context code) of an encountered error or exception in the ranking of search results. It also shows how *meta search* based idea can improve the performance of the search where our proposed approach outperforms two existing approaches from the literature in terms of all performance metrics. Our approach also performs significantly better than three traditional search engines especially in terms of *recall*, and its *precision* is comparable to Google, the best performing search engine. It should be noted that in the first study, we did not focus much on the development of fine-tuned search queries for the encountered exceptions, which might be a possible explanation for the comparable *precision* of our approach. Each search query was used to collect the top results from different search engines (for dynamic corpus development), and as we noted in the study, sometimes those queries failed to return the potential result links for the corresponding exceptions. In this chapter, we thus discuss our second study that proposes a novel approach for *context-aware* search query recommendation for programming errors and exceptions.

The rest of the chapter is organized as follows– Section 4.2 presents a motivating example and Section 4.3 explains the working methodologies, proposed metrics, and the ranking algorithms. Section 4.4 discusses the experimental design, results and validation details, Section 4.5 identifies the possible threats to validity, Section 4.6 focuses on the related works, and finally Section 4.7 summarizes the chapter with future works.

4.1 Introduction

A number of studies are conducted on supporting developers with errors and exceptions, and they can be grouped into two broad categories–*static analyzer* and *recommendation system*. The first group of studies [41, 49, 74] analyze exceptional control flows, program behaviours and other details in order to understand the exception handling structures of a software system. On the other hand, the recommendation systems in the latter group recommend different relevant items for a given exception to the developers such as web pages [43, 62, 69, 72], bug reports [52], solution history [54], and exception handling code examples [33]. Most of these recommendation systems generate search queries explicitly or implicitly by extracting necessary keywords from the technical details (e.g., stack trace) or the context (e.g., context code) of an encountered exception. However, none of them are specialized in search query recommendation, and their queries might

not be applicable for direct use in the traditional web search due to certain restrictions. First, the queries are often formatted with specific notations to meet the specifications of the corresponding recommender service module [33, 52], and they cannot be directly used as search queries. Second, the queries generated by those existing systems are often long (i.e., contain up to ten program tokens) [69, 70], and the traditional web search engines (e.g., Google, Bing and Yahoo) do not support long queries. Third, existing approaches [43, 70] analyze either the stack trace or the context code of an encountered exception in order to extract suitable tokens for a search query for the exception. They also consider either *degree of interest* [43] or *frequency* [69] of a token, and determine the appropriateness (i.e., suitability) of the token in the search query. Both the heuristic metrics are overly simple or not much effective in isolation, and thus, the chosen tokens might not represent the exception or its context in the search query properly. Our findings from the experiments in the first study (Section 3.3.6) also support this observation.

In this study, we propose a novel query recommendation approach that recommends suitable search queries for given programming errors and exceptions by exploiting their technical details and contexts. To the best of our knowledge, there exist no studies for such query recommendation. The proposed approach exploits stack trace and context code (i.e., a segment of the code that triggers the exception) of an encountered exception in the IDE, and adopts a popular graph-based term-weighting technique [34] (e.g., Google's PageRank) in addition to combining and refining existing heuristics (e.g., *degree of interest, frequency*) in order to choose suitable keywords for a search query. It recommends a ranked list of suitable queries within the IDE, which can be readily used by a developer for web search. The approach also overcomes certain limitations with the existing recommendation systems. First, each of the queries is recommended as a list of keywords chosen from the detailed context (e.g., stack traces, context code) of an exception, and it is readily applicable to keyword-based web search. Second, the recommended queries are generally of short-length, and our approach also provides query-length customization feature. Third, the proposed approach not only combines or refines the existing heuristics [43, 69, 70] of *token importance* in the context of the exception but also applies a popular graph-based term-weighting technique (i.e., widely used in information retrieval) in order to capture the importance (i.e., suitability) of a program token (e.g., class name and method name) for the search query. Fourth, the search query consists of only program artifacts such as exception name, class name or method name, and thus it could be possibly used with code search engines as well for relevant code examples.

We conduct experiments on the proposed approach using 50 exceptions and associated details (stack traces and context code) collected from our previous study [72] that recommends relevant web pages for programming errors and exceptions. We analyze exception message, stack trace and context code of each of the exceptions, and collect solution links from three popular search engines (Google, Yahoo and Bing) in order to develop an *oracle* (i.e., a gold-set of solutions). We then collect search results against the top ranked queries by our approach for each of the exceptions, and evaluate them using the *oracle*. The experiments show that our recommended queries return relevant web pages with 55.31% *mean-average precision* and 35.23% *recall*, and the results solve 80% of the exceptions, which are promising according to the relevant existing

Listing 4.1: Context Code of an Exception

```
44 FileInputStream fis = new FileInputStream(file);
45 ObjectInputStream ois = new ObjectInputStream(fis);
46 ArrayList<Record> currentList = new ArrayList<>();
47 // restore the number of objects
48 int size = ois.readInt();
49 // restore the objects
50 for (int i = 0; i < size; i++) {
51   Record current = (Record) ois.readObject();
52   currentList.add(current); }
```

Listing 4.2: Corresponding Stack Trace of the Exception from Listing 4.1

```
1 Exception in thread "main" java.io.EOFException
2 at java.io.ObjectInputStream$PeekInputStream.readFully(ObjectInputStream.java:2281)
3 at java.io.ObjectInputStream$BlockDataInputStream.readShort(ObjectInputStream.java
    :2750)
4 at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:780)
5 at java.io.ObjectInputStream.<init>(ObjectInputStream.java:280)
6 at HighScores.<init>(HighScores.java:45)
7 at HighScores.main(HighScores.java:151)
```

literature [43, 72]. We also compared our queries against traditional search queries and queries from three existing approaches for the same dataset, and found that the recommended queries by our proposed approach are more effective than others in terms of *mean average precision*, *recall*, and *percentage of exceptions solved*. Finally, in order to further validate the applicability of our queries, we conducted a user study with five participants (graduate research students), and found that the search queries provided by the participants matched with our recommended queries with a *pyramid score* of 0.84, which is also highly promising.

4.2 Motivating Example

Let us consider a problem solving scenario where a developer attempts to restore a list of serialized objects from a file, and encounters an *EOFException*. The exception is triggered from the context code in Listing 4.1, and the IDE reports the corresponding stack trace in Listing 4.2. The reported information by the IDE does not provide much insight into how to solve the exception, and the developer looks into web for relevant information. However, she faces several challenges in formulating the search query. First, the technical error message (e.g., Line 1 in Listing 4.2) is too generic, and the stack trace contains a lot of information. Neither the error message nor the trace information is a good candidate for search query. Second, although the stack trace helps her identify the source line (e.g., Line 45 in Listing 4.1) that triggers the exception, she needs more insightful analysis about the structure (e.g., hierarchical method calls) and behaviour of the exception as well as automatic supports (e.g., automatic query suggestion and completion) in order to be able to actually choose a suitable search query, which are not provided by either the IDE or any of the existing approaches. Third, the traditional search engines suggest query phrases during web search, which might be popular but not often appropriate enough for the programming problem at hand.

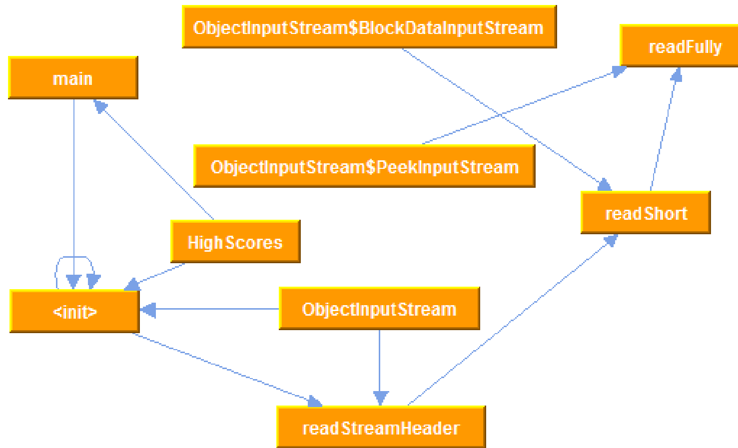

Figure 4.1: Stack Trace Token Graph of Stack Trace in Listing 4.2

Now let us consider the supports provided by our proposed approach in formulating search queries for the encountered exception. Listing 4.3 shows a ranked list of top five recommended queries by our approach. These queries can benefit the developer in the formulation of suitable queries from several perspectives. First, the queries are developed by carefully extracting (e.g., AST analysis, island parsing) suitable tokens from the stack trace in Listing 4.2 and context code in Listing 4.1. They then are recommended in the form of automatic suggestions within the IDE, which reduces the developer effort for query formulation. Second, each of the queries are represented as a list of tokens tuned for keyword-based search, and one can readily use them. Third, the approach visualizes the static relationships (i.e., between classes and methods) and call dependencies (i.e., method call sequences) found in the trace information as a *trace token graph* (Fig. 4.1), and demonstrates the importance of a token among other candidate tokens in terms of connectivity. In other words, the graph provides a high-level overview of the structure and the control flow of the encountered exception, which may help the developer in preparing customized search queries or even may facilitate the resolution of the exception without a web search.

Listing 4.3: Recommended Queries for Context Code in Listing 4.1 and Stack Trace in Listing 4.2

```
1 java.io.EOFException ObjectInputStream readShort readFully
2 java.io.EOFException readStreamHeader readShort readFully
3 java.io.EOFException readShort readFully PeekInputStream
4 java.io.EOFException ObjectInputStream readStreamHeader readFully
5 java.io.EOFException ObjectInputStream readFully PeekInputStream
```

4.3 Proposed Approach

In this section, we discuss the working methodology of our approach, our proposed metrics and ranking algorithms for query tokens, and query formulation, ranking and recommendation techniques.

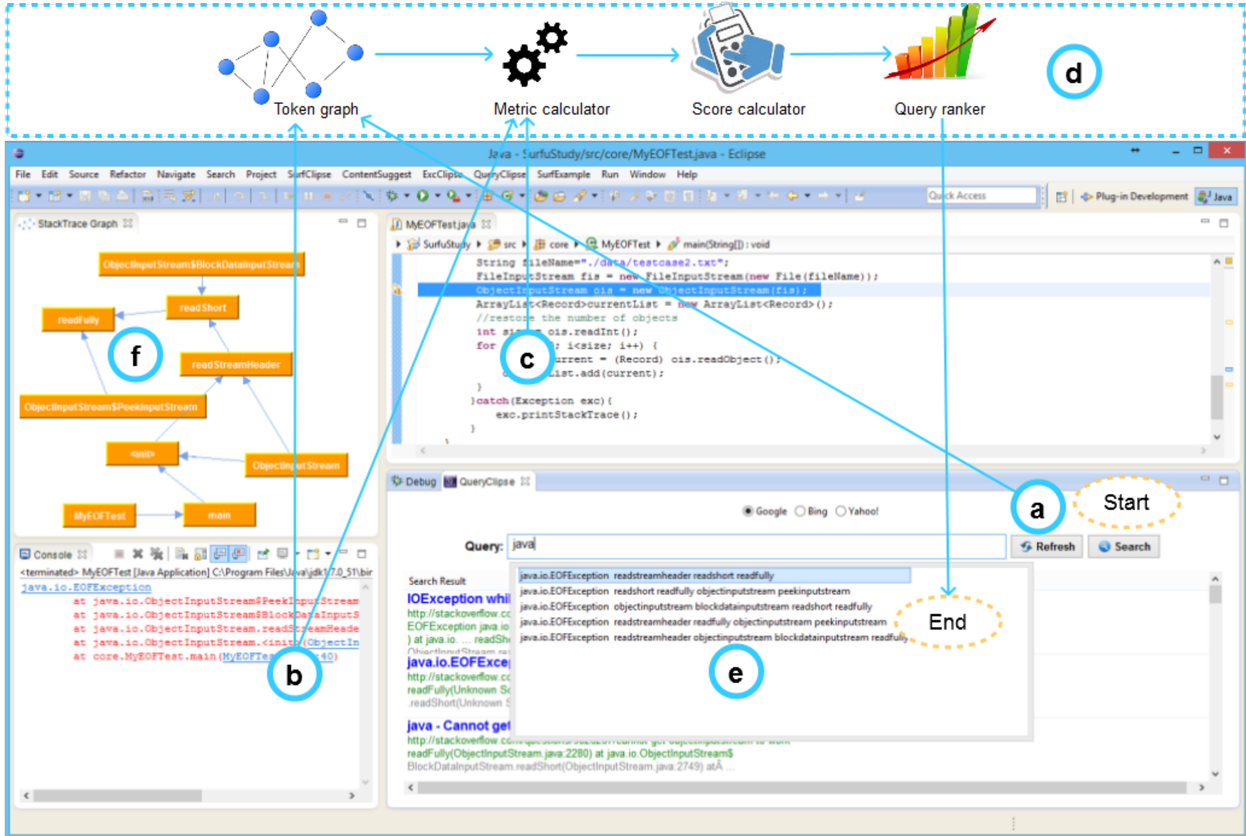

Figure 4.2: Schematic Diagram of Proposed Approach (QueryClipse)

4.3.1 Methodology

The schematic diagram of our proposed approach in Fig. 4.2 shows different steps involved in search query recommendation. We package our recommendation service as an Eclipse plugin, *QueryClipse* [23]. Once an exception occurs and a developer requests for search queries (Fig. 4.2-(a)), technical details (e.g., error message and stack trace) (Fig. 4.2-(b)) and context code (Fig. 4.2-(c)) of the exception are collected, and the trace information is analyzed to develop a token graph (Fig. 4.2-(d)). The graph encodes the connectivity among different trace tokens based on implied static relationships (e.g., class-to-method relations, *ObjectInputStream*-to-*readStreamHeader*) and call dependencies (i.e., sequence of calls) among different methods in the stack trace (Fig. 4.2-(b)). The computation module (Fig. 4.2-(d)) then analyzes the constructed graph, exception details and the context code in order to derive different metrics and heuristic measures, which are used to calculate the score (i.e., relative importance) of each of the individual tokens in the graph. It then chooses the top five tokens, and develops different combinations of tokens, which are ranked (i.e., based on individual token score), and then recommended to the developer in the IDE (Fig. 4.2-(e)). The proposed approach also visualizes the constructed token graph (e.g., Fig. 4.2-(f)), and the developer can prepare customized search queries by choosing appropriate tokens from the graph for the exception.

4.3.2 Proposed Metrics

To the best of our knowledge, there exist no studies that focus on the search query recommendation for programming errors and exceptions. Existing studies [43, 69, 70] that recommend relevant posts from StackOverflow[1] Q & A site for a given programming exception or any other problem, generate search queries implicitly. They consider either *frequency* of a program token in the context code [69, 70] or *proximity* of a trace token to the location of exception in the stack trace [43], and use the top ranked tokens to develop a search query. However, both heuristic measures are overly simple and are derived from the partial context (i.e., only either stack trace or context code) of the exception, and thus they may not be very effective for query formulation. In our study, we refine those heuristics, adapt another metric from information retrieval domain, and then combine all three measures into a compound metric– *token score*. This section discusses our proposed metrics that are used to determine the suitability or appropriateness of a trace token in the search query for the exception.

Trace Token Rank (TTR)

Existing studies on information retrieval [31, 34, 56] often use a graph-based term-weighting technique, where they consider a text document as a network (i.e., graph) of interdependent (i.e., for meaning) terms. The dependencies among the terms are determined based on their co-occurrences in the document. In our research, we consider the stack trace (e.g., Listing 4.2) of an exception as such a document, and exploit the dependencies and static relationships among the trace tokens in order to determine their applicability or importance in the search query. However, we adopt a different approach other than the approach of co-occurrence in determining the relationships and the dependencies among the tokens. Nguyen et al. [67] propose a graph-based representation for API usage in a code segment, where they show the static relationships (e.g., attribute to class relationships) and dependencies (e.g., data dependency and temporal usage order) among different program elements (e.g., API objects, methods) in the graph. A stack trace generally contains a series of method call references, which also encode such static relationships and dependency (e.g., control flow) information. We thus adapt the graph-based representation by Nguyen et al. for the stack trace, and develop a token graph (e.g., Fig. 4.1) by extracting suitable tokens such as class name and method name from the trace information. We also exploit the static relationships between Java classes and their methods as well as the caller-callee relationships among the methods in order to connect the tokens in the graph. In the graph (e.g., Fig. 4.1), the tokens are connected with inbound and outbound links, and we apply the graph-based term-weighting technique by Blanco and Lioma [34], an adaptive version of PageRank algorithm (Section 2.8), in order to determine the *Trace Token Rank (TTR)* as follows:

$$TTR(T_i) = (1-d) + d \times \sum_{k \epsilon In(T_i)} \left(\frac{TTR(T_k)}{|Out(T_k)|}\right) \tag{4.1}$$

[1] http://stackoverflow.com

Here, $In(T_i)$ refers to the list of tokens to which token T_i is connected through inbound links, $Out(T_i)$ refers to such a list to which T_i is connected with outbound links, and d is the damping factor. In the context of PageRank algorithm that models browsing behaviours of the web surfers, damping factor is the probability of randomly clicking a page by a surfer in the web for browsing. In our research, we use it as the probability of a token to be selected as a keyword in the search query. We use an iterative version of PageRank score calculation with a limit of 100 iterations, and collect *token rank score* for each of the tokens in the token graph. In order to simplify the final score calculation, we also normalize the *token rank* of each of the tokens.

Degree of Interest (DOI)

Cordeiro et al. [43] propose a heuristic measure, *Degree of Interest*, in order to estimate the proximity of a method call reference in the stack trace to the location of corresponding exception in the code. They apply the measure to each of the tokens in each call reference, and then use the tokens along with those measures for query formulation and for determining structural similarity between two stack traces. In our research, we leverage this heuristic measure in order to determine the applicability of a stack trace token in the search query. The idea is– the closer a token is to the location of exception, the more applicable it is for a search query. If a trace block contains N method call references, then *Degree of Interest*, S_{doi}, for each of the call references can be estimated as follows:

$$S_{doi} = 1 - \frac{n_i - 1}{N} \tag{4.2}$$

Here, n_i represents the position of a method call reference in stack trace. We consider each of the tokens in the token graph (e.g., Fig. 4.1) as a query token candidate, and collect the *degree of interest* from its corresponding method call reference. In case of the presence of a single token in multiple call references, we average the measures. The proximity measure values from zero to one, where zero indicates that the token (or its call reference) is far away from the location of exception, and one means that the token is most likely in the reference that triggers the exception.

Trace Token Frequency (TTF)

Ponzanelli et al. [69] analyze the source code under development in the IDE, and use the ten most frequent program tokens (e.g., class name, method name) as a search query in order to collect relevant StackOverflow posts for the programming task at hand. In this research, we consider such source code tokens that are found in the trace information. The idea is that the reported classes and methods in the stack trace are more likely responsible for the encountered exception, and an appropriate search query for the exception should include them. However, the stack trace often contains a number of tokens, and the most frequent ones in the context code should be privileged. In our research, we thus consider each of the tokens in the token graph, and calculate their frequency in the context code (e.g., Listing 4.1) of the encountered exception. In case of Java classes, we consider the number of instances, and in case of methods, we consider the invocation frequency.

We then assign a normalized frequency score, $TTF(T_i)$, to each of the tokens as follows:

$$TTF(T_i) = \frac{F_i}{max(F_i)} \tag{4.3}$$

Here F_i denotes the actual frequency of a trace token in the context code, and $max(F_i)$ refers to the maximum frequency among those of all tokens in the token graph.

4.3.3 Token Score Calculation

In this research, we consider three aspects–*token rank, proximity to the location of exception* and *token frequency* for the score calculation (i.e., determining importance) of each of the chosen program tokens (e.g., class name and method name) from the stack trace. *Token rank* shows the applicability of a Java class name or a method name as a query token by exploiting its implied static relationships and call dependency information in the stack trace. On the other hand, the rest two heuristic measures show the importance of a trace token in terms of its location in the stack trace and frequency in the context code respectively. In order to determine the final score of a trace token, we simply combine all three normalized measures as follows:

$$S(T_i) = \alpha \times TTR(T_i) + \beta \times DOI(T_i) + \gamma \times TTF(T_i) \tag{4.4}$$

Here, TTR, DOI and TTF are *trace token rank, degree of interest* and *trace token frequency* of a token respectively, and α, β and γ are the relative weights of the corresponding metrics. We consider a heuristic value of 0.90 for α, 1.00 for β, and 0.50 for γ. We choose these heuristic weights based on our extensive and iterative controlled experiments with a subset of 20 exceptions and their corresponding search queries, manual analysis on the experimental results, discussion among the authors, and also some helpful ideas from the existing studies [43, 72]. While these heuristic values might seem a bit arbitrary, we find the combination to be the best in our experiments to represent the relative importance of different aspects of the final score for a trace token.

4.3.4 Query Ranking & Recommendation

Once the final scores of the tokens are calculated, we choose a list of the top scored five tokens (i.e., motivated by the approach of Cordeiro et al. [43]). It should be noted that we do not include the insignificant tokens such as *main, init* or *(init)* in the list, and the list only contains different important *class name* and *method name* tokens. We then develop different combinations of tokens choosing a fixed number of tokens each time from the list. We experimented with different sizes for each token combination, and finally choose a size of three (i.e., fixed number) for the combinations. Each of these combinations of tokens is a candidate query, and we calculate the score of the query based on its corresponding token scores. We then rank each of the candidate queries, and recommend the top five candidates. It should be noted that name and technical error message of the encountered exception are prepended to each of the candidates in order to develop the final queries. For example, Listing 4.3 shows the top ranked five search queries for the showcase exception with stack trace in Listing 4.2 and context code in Listing 4.1.

4.4 Experimental Design, Results and Validation

In this section, we discuss the detailed design of the conducted experiments, analyze the results and validate them against three existing approaches. We also study the applicability of the recommended search queries by our approach using a user study with five participants.

4.4.1 Dataset and Tools

In our experiments, we use 50 exceptions, their technical details (e.g., stack traces) and context information (e.g., context code). Most of them are collected from our previous work [72] on the recommendation of web pages for programming errors and exceptions, and a few of them are collected from different online sources such as pastebin [20] and StackOverflow. We collect the most appropriate solutions for those exceptions with the help of four available search engines– Google, Bing, Yahoo and Ask, and develop an *oracle*. Given that selection of solutions for an exception is a subjective approach, we validate the solutions in the oracle with the help of peers. All the data used for experiments can be found online [22]. During score calculation of trace tokens, the stack trace of each exception is analyzed for *trace token graph*, and we use a popular Java library, JGraphT[2] for graph-based analysis and visualization. We also analyze the context code of the exception in order to detect the occurrences of the trace tokens. In the analysis of the code, we use an AST-based parser, JavaParser[3] (for compilable code) and an island parser (for non-compilable code) [72] in order to extract class object instantiation and method call references.

4.4.2 Existing Query Recommender

We investigate the existing search query recommendation services. Most of the traditional keyword-based search engines (e.g., Google, Bing and Yahoo) support search query suggestions based on past queries[4] made by other users. The support sometimes helps a developer to choose an appropriate search query for the encountered programming error or exception through several attempts; however, it comes with two serious limitations. First, it only can recommend useful queries for the errors and exceptions which are widely discussed and searched over the Internet in the past. Second, both an IDE and a search engine work in different context from each other, and the search engine is not generally aware of the context (i.e., surroundings, circumstances) of the encountered exceptions in the IDE. Thus, it recommends queries or search results irrespective of the problem context of interest, and a developer often faces difficulties in choosing an appropriate search query for the encountered exception. In other words, she is responsible for carefully representing the context of the problem to the search engine in order to get either relevant query suggestions or relevant search results.

[2]http://jgrapht.org
[3]http://code.google.com/p/javaparser
[4]http://www.google.com/support/enterprise/static/gsa/docs/admin/70/gsa_doc_set/xml_reference/query_suggestion.html

4.4.3 Performance Metrics

Our proposed approach profoundly aligns with the research areas of information retrieval and recommendation systems. In order to evaluate the effectiveness of our recommended queries, we conduct an extensive web search and a user study using them, and apply the following performance metrics for evaluation:

Mean Average Precision at K (MAPK): *Precision at K* calculates *precision* at the occurrence of every relevant result in the ranked list. *Average Precision at K (APK)* averages the *precision at K* for all relevant results in the list for a search query. *Mean Average Precision* is the mean of *average precision at K* for all queries in the dataset.

$$APK = \frac{\sum_{k=1}^{D} P_k \times rel_k}{|RR|} \tag{4.5}$$

$$MAPK = \frac{\sum_{q \epsilon Q} APK(q)}{|Q|} \tag{4.6}$$

Here, rel_k denotes the relevance function of kth result in the ranked list, P_k denotes the precision at k^{th} result, and D refers to number of total results. RR is the set of relevant results for a query, and Q is the set of all queries in the dataset.

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

Pyramid Score (PS): Haiduc et al. [53] use *Pyramid score* [66] in order to compare automatic summaries of source code against developer provided summaries. In our research, we use this metric to compare the recommended queries by our approach against the user provided search queries for an exception in the user study (Section 4.4.6). Suppose $N = 5$ participants choose a list of total $M = 10$ tokens in their search queries, and the recommended query contains $K = 5$ tokens. Now we sum up the frequency of each of the recommended tokens in the user queries, and also identify the top most K frequent tokens in those queries.

Table 4.1: Pyramid Score Calculation

	exception	**util**	**concurrent**	**modification**	**arraylist**	**abstractlist**	**java**	**thread**	**main**	**next**
P1		x	x	x		x		x		
P2	x		x	x	x			x		
P3		x	x	x	x				x	
P4	x	x	x			x			x	
P5			x	x	x	x				x
RQ[1]	x(2)		x(5)	x(4)	x(3)					x(1)

RQ=Recommended Query, **Pyramid score**=(5+4+3+2+1)/(5+4+3+3+3)=15/18=0.83

For example, in Table 4.1, *concurrent, modification, exception, arraylist* and *next* are the recommended query tokens by our approach, and the sum of their frequencies is 15. On the other hand, the most frequent five tokens– *concurrent, modification, util, arraylist* and *abstractlist* in the user queries provide such a sum of 18, and *Pyramid score* is calculated as a ratio of these two summations, which is 0.83 in this case. The

score values from zero to one, where one indicates that the recommended query is comparable to fine-tuned user queries, and zero indicates that the recommended query does not match with the user queries at all. A pyramid score around 0.5 indicates that the query matches with the fine-tuned user queries moderately, and thus may not be promising.

4.4.4 Experimental Results

We conduct experiments with 50 exceptions and their context details (stack traces and context code), where the proposed approach analyzes the stack trace and the context code of each of the exceptions, and returns a list of ranked search queries. We choose the top-ranked five queries, and collect the top 20 search results from each of the three most popular web search engines (i.e., according to Alexa[5] ranks)– Google, Yahoo and Bing by accessing their API endpoints. The search results are then analyzed and evaluated using the *oracle*, and Table 4.2 and Table 4.3 report the evaluation details.

Table 4.2 reports the findings of our experiments on the proposed approach, where all three metrics– *Degree of Interest, Trace Token Rank* and *Trace Token Frequency* are considered both in isolation as well as in combination. We note that *Degree of Interest* and *Trace Token Frequency* metrics perform quite well in terms of *percentage of exceptions solved (PTCS)* (e.g., 78.00%) and *mean average precision (MAPK)* (e.g., 57.00%) respectively for all search engines except Google. *Trace Token Rank* performs moderately (e.g., 76.00% and 52.65% respectively) in terms of both metrics. When any two metrics are combined, we found the combination–{*Degree of Interest* and *Trace Token Rank*} performs the best in terms of *recall* (e.g., 35.23%) and *percentage of exceptions solved* (e.g., 78.00%), and moderate in terms of *precision*. On the other hand, the combination-{*Degree of Interest* and *Trace Token Frequency*} provides a relatively better *mean average precision* (e.g., 53.49%) while it solves almost equal number of exceptions. However, when we consider all three proposed metrics (Section 4.3.2) in combination, our approach performs the best in terms of almost all performance metrics with Bing and Yahoo search engines. For example, Yahoo solves 80% of the exceptions with 35.23% *recall* and a maxiumum of 55.31% *precision*. We note the relatively lower performance by our approach with Google search for both isolated and combined metrics, and we investigate the issue later in the chapter.

Our proposed approach recommends a ranked list of search queries instead of a single query for an exception of interest. Table 4.3 reports the findings of our investigation into the effectiveness of the ranking in search queries. We choose the top ranked five queries for an exception, and conduct web search using three search engines. Given that conducting such search with all exceptions in the dataset is a non-trivial task, we chose ten exceptions (or stack traces) with trivial error messages (e.g., contain exception names only). The idea is to examine the effectiveness of our queries for the exceptions that provide a developer with limited opportunities to develop search queries. From Table 4.3, we note that Bing performs significantly well in terms of all three performance metrics–*mean average precision* (e.g., 74.26%), *recall* (e.g., 41.18%)

[5]http://www.alexa.com/topsites, visited on June 2014

Table 4.2: Results of Experiments on Different Aspects for Token Importance

Score Combination	TE	Metric	Google		Bing		Yahoo	
			Top 10	Top 20	Top 10	Top 20	Top 10	Top 20
Degree of Interest (DOI)	50	MAPK	39.84%	39.84%	49.48%	47.90%	**50.12%**	48.56%
		R	17.61%	17.61%	27.84%	30.68%	30.68%	**32.95%**
		PTCS	**52.00%**	52.00%	72.00%	74.00%	78.00%	**78.00%**
Trace Token Rank (TTR)	50	MAPK	36.46%	36.46%	52.40%	**52.65%**	47.69%	46.62%
		R	15.34%	15.34%	28.97%	30.11%	32.95%	**34.65%**
		PTCS	46.00%	46.00%	70.00%	74.00%	74.00%	**76.00%**
Trace Token Frequency (TTF)	50	MAPK	37.57%	37.57%	**57.00%**	**55.75%**	**55.47%**	**55.46%**
		R	14.77%	14.77%	22.16%	23.29%	25.57%	**25.57%**
		PTCS	46.00%	46.00%	66.00%	66.00%	68.00%	**68.00%**
{DOI, TTR}	50	MAPK	36.36%	36.36%	49.24%	49.04%	**51.70%**	51.61%
		R	15.91%	15.91%	27.84%	**30.68%**	34.09%	**35.23%**
		PTCS	48.00%	48.00%	70.00%	**76.00%**	76.00%	**78.00%**
{TTR, TTF}	50	MAPK	**38.23%**	38.23%	50.18%	50.09%	45.46%	44.60%
		R	15.34%	15.34%	29.55%	31.25%	30.68%	32.39%
		PTCS	46.00%	46.00%	70.00%	74.00%	68.00%	70.00%
{DOI, TTF}	50	MAPK	37.26%	37.26%	49.53%	48.23%	**53.49%**	51.35%
		R	17.61%	17.61%	27.84%	30.11%	30.68%	32.95%
		PTCS	**50.00%**	50.00%	72.00%	74.00%	78.00%	**78.00%**
{DOI, TTR, TTF}	50	MAPK	34.06%	34.06%	51.85%	50.44%	**55.31%**	**53.40%**
		R	13.64%	13.64%	27.84%	31.25%	31.82%	**35.23%**
		PTCS	42.00%	**42.00%**	72.00%	**76.00%**	76.00%	**80.00%**

TE=Total exceptions in the experiment, **PTCS**=Percentage of the exceptions solved.

and *percentage of exceptions solved* (e.g., 100%) for the *Rank II* queries, and Yahoo does almost the same for the *Rank III* queries. Both of them also perform comparatively in terms of all three performance metrics for the top ranked (i.e., *Rank I*) queries. On the other hand, both search engines perform relatively poor for *Rank IV* and *Rank V* queries. Thus the findings indicate that the most effectives queries can be found in the top three positions of the ranked list by our approach. It is also observed that the queries of a particular rank may not be equally effective for different search engines, which validates our idea of recommending a list of queries. The reduced effectiveness of the queries in the bottom of the list also indicates that the ranking of the queries might be quite meaningful, and this finding is also partially validated by the results from our conducted user study (Section 4.4.6).

4.4.5 Comparison with Existing Approaches

In order to validate performance of the recommended queries by our approach, we compare them with traditional search queries and the queries from three existing approaches–Cordeiro et al. [43], Ponzanelli

Table 4.3: Results of Experiments on Different Ranked Queries

Query Rank	TE	Metric	Google		Bing		Yahoo	
			Top 10	**Top 20**	**Top 10**	**Top 20**	**Top 10**	**Top 20**
Rank I	10	MAPK	17.43%	17.43%	67.92%	67.92%	**73.10%**	68.86%
		R	11.76%	11.76%	38.24%	38.24%	38.24%	**41.18%**
		PTCS	**30.00%**	30.00%	90.00%	90.00%	90.00%	**90.00%**
Rank II	10	MAPK	20.00%	20.00%	74.26%	**74.26%**	63.33%	63.33%
		R	5.88%	5.88%	41.18%	**41.18%**	38.24%	**38.24%**
		PTCS	20.00%	20.00%	100.00%	**100.00%**	100.00%	**100.00%**
Rank III	10	MAPK	27.00%	27.00%	**71.43%**	**71.43%**	**68.65%**	**69.56%**
		R	11.76%	11.76%	35.29%	35.29%	41.18%	**44.18%**
		PTCS	40.00%	40.00%	90.00%	90.00%	90.00%	**100.00%**
Rank IV	10	MAPK	33.67%	33.67%	61.42%	61.43%	**66.76%**	66.76%
		R	14.71%	14.71%	29.41%	**29.41%**	35.29%	**35.29%**
		PTCS	50.00%	50.00%	80.00%	**80.00%**	90.00%	**90.00%**
Rank V	10	MAPK	27.00%	27.00%	70.00%	68.02%	67.83%	67.83%
		R	14.71%	14.71%	29.41%	32.35%	38.23%	38.23%
		PTCS	50.00%	50.00%	90.00%	90.00%	90.00%	90.00%

TE=Total exceptions in the experiment, **PTCS**=Percentage of total exceptions solved

Table 4.4: Comparison with Queries from Traditional and Existing Approaches

Query Recommender	TE	Metric	Google		Bing		Yahoo	
			Top 10	**Top 20**	**Top 10**	**Top 20**	**Top 10**	**Top 20**
Traditional Approach (Exception message only)	50	MAPK	**38.97%**	38.97%	**44.11%**	43.82%	43.18%	43.18%
		R	19.88%	19.88%	24.43%	26.14%	25.00%	25.00%
		PTCS	52.00%	52.00%	58.00%	**60.00%**	56.00%	**56.00%**
Cordeiro et al. [43]	50	MAPK	**21.33%**	21.17%	**19.22%**	19.22%	15.94%	16.60%
		R	10.80%	11.93%	11.93%	13.07%	10.80%	13.06%
		PTCS	36.00%	**38.00%**	34.00%	**36.00%**	32.00%	**40.00%**
Ponzanelli et al. [70]	50	MAPK	**14.36%**	14.36%	**30.27%**	29.98%	28.12%	28.12%
		R	9.09%	9.09%	12.50%	13.07%	12.50%	12.50%
		PTCS	24.00%	**24.00%**	38.00%	**38.00%**	38.00%	**38.00%**
Rahman et al. [72]	50	MAPK	**31.96%**	31.96%	**54.93%**	53.73%	54.29%	52.96%
		R	13.64%	13.64%	29.55%	31.82%	26.14%	28.41%
		PTCS	40.00%	**40.00%**	74.00%	**76.00%**	68.00%	**70.00%**
Proposed Approach (Recommended queries)	50	MAPK	34.06%	34.06%	51.85%	50.44%	**55.31%**	**53.40%**
		R	13.64%	13.64%	27.84%	31.25%	31.82%	**35.23%**
		PTCS	42.00%	**42.00%**	72.00%	**76.00%**	76.00%	**80.00%**

TE=Total exceptions in the experiment, **PTCS**=Percentage of total exceptions solved

et al. [69] and Rahman et al. [72]. It should be noted that these approaches do not focus on search query recommendation although they generate queries for search result collection. Table 4.4 reports the findings from our comparative studies.

Software developers often copy the technical error message (e.g., first line in the stack trace in Listing 4.2) of an encountered exception, and perform web search. In our comparative study, we consider such message from the stack trace as a traditional search query for each exception in the dataset, and compare them with our recommended queries. From Table 4.4, we note that our queries perform significantly better than traditional search queries in terms of all three performance metrics with especially Bing and Yahoo. For example, the traditional queries return results from any of the search engines with a maximum of 44.11% *mean average precision* and 26.14% *recall*, and the results solve only 60% of the exceptions. On the other hand, the recommended queries by our approach return results with a maximum of 55.31% *mean average precision* and 35.23% *recall*, and the results solve a maximum of 80% of the exceptions.

Cordeiro et al. [43] propose an IDE-based StackOverflow(SO) post recommendation approach, where they analyze tokens from the stack trace of an encountered exception in order to develop a search query. Their query development technique is partially similar to ours in the sense that they also consider the *Degree of Interest* metric. However, they simply capture all the tokens from a method call reference in the stack trace and treat them with equal importance. On the other hand, we carefully choose the *class name* and *method name* tokens from the reference, determine their relative importance using a graph-based term weighting approach, and also identify their occurrences in the context code. The idea is to prepare a search query that captures required information both from the stack trace and the context code. From Table 4.4, we note that the search queries by Cordeiro et al. can return results with a maximum of 21.33% *mean average precision* and 13.06% *recall*, and the results can solve only 40% of the exceptions, which are significantly poor compared to those of our queries.

Ponzanelli et al. [69] propose another IDE-based SO post recommendation approach that analyzes code under development in the IDE, and develops a search query. The approach is not intended for recommending SO posts about programming errors and exceptions; however, it follows a similar technique to ours in identifying tokens from the code. They analyze the code and extract the ten most frequent tokens containing *class name* and *method name* for a search query. On the other hand, we adopt a more selective strategy in this regard besides choosing other metrics. We choose only such source code tokens (e.g., class name, method name) those are found in the stack trace. The idea is that the reported classes and methods in the stack trace are more likely responsible for the encountered exception, and they should be preferred over other tokens for the search query. From Table 4.4, we note that the search queries by Ponzanelli et al. [69] can return results with a maximum of 30.27% *mean average precision* and 13.07% *recall*, and the results can solve at most 38% of the exceptions, which are relatively poor compared to those of our queries. The findings also indicate that search queries based on highly simplified heuristics (e.g., frequency) are not much effective.

In our previous work– *SurfClipse* [72], in order to develop a search query, we chose five tokens with the

highest *Degree of Interest* from the stack trace and five most frequent tokens from the context code of an encountered exception. Thus tokens from the context code were not chosen carefully, and the importance (i.e., suitability, appropriateness) of each individual token in the search query was not properly determined. In some cases, the generated queries were also too long to return results from the traditional search engines. On the other hand, the proposed approach in this study analyzes three important aspects–*token rank, token frequency* and *proximity to the location of exception* of each token in the token graph in order to determine its applicability in a search query. More importantly, we adapt a *graph-based term-weighting technique* of *information retrieval* domain for the stack trace of a programming exception, and estimate the relative importance of each token in an interconnected network of tokens (e.g., Fig. 4.1). From Table 4.4, we note that the search queries from our previous approach [72] return results with a maximum of 54.93% *mean average precision* and 31.82% *recall*, and the results solve at most 76% of the exceptions. The findings indicate that our previous approach [72] is quite comparable; however, the proposed approach still performs relatively better in terms of all three performance metrics.

In our experiments, we note that all search queries–*recommended, traditional* and *from existing approaches* perform relatively poorly with the Google search engine compared to with Bing and Yahoo. We investigate the issue and observe a remarkable scenario. Google follows a strict limit (e.g., 128 characters) in the query length [10], and does not return results for the queries that exceed that limit. However, as we noticed, Bing and Yahoo return results for the same queries, which might be a possible explanation for the relatively low performance of all the queries with Google. All the queries in our experiment contain multiple words, contain programming keywords and thus are complex in nature. Two existing studies [59, 81] show that Google performs relatively poor with complex multi-word queries compared to Yahoo, and Bing, and they also support our observation.

We also investigate the ten exceptions for which our recommended queries did not return any solution web link with any of the search engines. We analyze the search queries, and found two cases. First, most of those exceptions are too common (e.g., *FileNotFoundException, NumberFormatException*) and their queries (by our approach) are too generalized to retrieve the expected solutions within the top 20 positions. Second, we found four queries (for four exceptions) of excessive length due to long error messages from the stack traces, and no search engines did return results for them.

4.4.6 Conducted User Study

In order to validate the applicability of the search queries by our approach, we conducted a user study with five participants (i.e., graduate research students of Software Research Lab, University of Saskatchewan). We chose five exceptions from the dataset that contain little or no technical explanation about the cause of exception in the stack trace. The idea was to reduce the bias of using *error message only* as a search query and to avoid long query. We sent the stack trace and the context code of each exception to the participants, and asked them to report the most suitable search query for the exception, which returns solutions from any

Table 4.5: Comparison with User Provided Queries for Exceptions

Query No.	1	2	3	4	5	APS	MAPS
PS (Rank I)	0.75	0.89	1.00	1.00	1.00	0.93	0.84
PS (Rank II)	0.67	0.72	0.93	1.00	0.63	0.79	
PS (Rank III)	0.67	0.72	1.00	0.93	0.63	0.79	

PS=Pyramid score, **APS**=Average Pyramid Score

MAPS=Mean Average Pyramid Score

of the three search engines– Google, Bing and Yahoo. In order to help them with solution verification, we also provided the solution links from our developed *oracle*. Once we collected the queries from the participants, we compared the recommended queries by our approach with them. We use *Pyramid score* for the comparison, where the metric determines the extent to which an auto-generated (i.e., recommended) query resembles with the manually prepared search queries by the participants for the same exception. Table 4.5 reports the findings from the comparative analysis.

From Table 4.5, we note that each of top three recommended queries by our approach matches significantly with the queries from the participants. For example, *Rank I* queries returns an *average pyramid score* of 0.93, which is highly promising according to relevant literature [53, 66]. We also get higher score for Rank II and Rank III queries. These findings indicate that the recommended queries are quite similar to the best judged queries of the participants, and thus they are highly applicable for automated recommendation.

4.5 Threats to Validity

In our research, we identify several issues worthy of discussion. First, in order to develop a search query, we choose the best combination of program tokens (e.g., *class name, method name*) extracted both from the stack trace and the context code of an exception, and then add *exception name* and *error message* to the combination. Thus the proposed approach might recommend *long queries* in the case where the *error message* itself is lengthy. While describing a technical issue, the *error message* often contains different items such as *file name, URL* and *long package name* of a class associated with the exception. In order to mitigate the threat, we parse such items and remove them from the message. We also use only three tokens from the context of the exception in the search query in order to keep it short.

Second, the proposed approach may not always return such search queries which are friendly to the search engines, as it deals with complex program tokens from the stack trace and the context code of an exception. The non-friendly (i.e., not fine-tuned for a web search engine) search queries often fail to return enough results which is a potential threat to our approach. In order to mitigate the threat, we not only discard insignificant (e.g., *main, init, run* method tokens) and duplicate tokens (i.e., one token contains another token) from the search query but also normalize (e.g., extracting out the root token, converting *invoke0* to *invoke*) each of the tokens in the query.

Third, during the experiments, we note that the search engines evolve rapidly within days and weeks, and the results from a search engine may vary over time for the same query. Therefore, the statistics from the experiments with the search engines are very likely to change. We recommend a ranked list of similar queries for an exception rather than a single best query so that the developers can adapt with the changed behaviour of the search engines and still can retrieve the solution links.

4.6 Related Work

There exist a number of studies on supporting developers with programming errors and exceptions, and they can be grouped into two broad categories–*static analyzer* and *recommendation system*. The first group of studies analyze and visualize exceptional control flows, program behaviours and other details in order to understand the exception handling structures of a software system [41, 49, 74]. On the other hand, the recommendation systems in the latter group recommend different relevant items to the developers such as web pages [43, 62, 70, 72], bug reports [52], solution history [54], and exception handling code examples [33]. Most of these recommendation systems generate search query implicitly by extracting necessary keywords from the technical details of an encountered error or exception. However, those queries are often formatted with specific notations to meet the specifications of the corresponding recommender service module [33, 52], and they cannot be directly used as queries for traditional web search. Our work falls into the second category, and to our knowledge, this is the first attempt to recommend search queries for especially programming errors and exceptions.

Cordeiro et al. [43] propose a recommendation system that analyzes the stack trace of an encountered exception in the IDE, and recommends relevant StackOverflow links. Ponzanelli et al. [70] propose another recommendation system that analyzes the code under development in the IDE, and recommends relevant StackOverflow links. While these two approaches exploit StackOverflow data source only, our earlier work [72] proposes a meta search engine that collects results from four search APIs–*Google, Bing, Yahoo* and *StackOverflow* about an encountered exception in the IDE, analyzes and ranks the results, and then recommends to the developers. All three approaches above are closely related to our work since each of them generates search queries implicitly following a similar approach like ours. We compared our recommended queries for each exception in the dataset with theirs, and found that our queries are comparatively more effective. Our queries return results with relatively better *precision* and *recall* and they return correctly for more exceptions of the dataset. For a detailed comparison, readers are referred to Section 4.4.5.

There exist other studies which are not directly compared with our approach although they are related to our work. Gu et al. [52] propose an IDE-based approach that returns the bug reports discussing errors and exceptions similar to the ones encountered in the IDE. They analyze stack trace or execution trace reported in the bug description and use Bug Query Language (BQL) to retrieve and recommend relevant bug reports. Hartmann et al. [54] propose a social recommender system, *HelpMeOut*, for programming errors

and exceptions that captures problem solving practices (e.g., a list of code fixes to solve an exception) of the peers in the community, and recommends them in the IDE on the occurrence of similar exception. Marceau et al. [62] conduct a series of studies with student programmers, and explore and analyze interactions of the programmers with different error or exception messages in the IDE. They report that those messages often fail to convey the technical problems to the programmers. While the above approaches focus on recommending different programming artifacts such as bug reports, code examples or simplified error messages, our proposed approach recommends a list of context-sensitive and ready-to-use search queries by carefully analyzing the technical details (e.g., stack trace) and associated context (e.g., context code) of an encountered exception.

Vahabi et al. [82] propose an orthogonal query recommendation approach that offers query reformulation service. Their approach recommends alternative queries those are syntactically different but semantically similar to a previous unsuccessful query. Ensan et al. [45] propose a graph-based query recommendation approach that recommends query terms sequentially based on the past behaviours of other users of the search engine. While these studies are closely related to our work, they basically analyze the search engine usage data for recommendation, and do not consider the technical details of an encountered exception. Thus they might recommend popular search queries (i.e., as traditional search engines do); however, it cannot be guaranteed that those queries would be relevant and useful for the encountered exception in the IDE.

4.7 Summary

In order to complement our meta search engine (Chapter 3) in search query generation and recommendation, in this chapter, we propose a novel approach for context-aware search query recommendation for programming errors and exceptions. The approach exploits *token rank*, *frequency* and *proximity* of a stack trace token to exception location in order to determine the suitability of the token in a search query. We conduct experiments with 50 exceptions and their context details (e.g., stack traces, context code), collect search results from three popular search engines using the recommended queries, and evaluate them using the *oracle*. Our queries return results with 55.31% *mean average precision* and 35.23% *recall*, and the results solve 80% of the exceptions, which are promising. The recommended queries are also found to be more effective than the traditional queries (e.g., exception message) and the queries from three existing approaches in terms of all performance metrics. The conducted user study also shows that our queries are highly suitable for recommendation. While both the meta search engine in Chapter 3 and the proposed query recommender in this chapter are significant additions to the developers' tool chain for automated support in problem solving, the tool support can be further extended to post-search content analysis. In the next chapter (Chapter 5), we thus propose a novel content recommendation approach that extracts and recommends the most relevant section from a given web page for a programming exception of interest.

Chapter 5

Context-Aware Content Suggestion from a Web Page

Our previous two studies (Chapter 3 and Chapter 4) are on the recommendation of suitable search queries and relevant web pages for programming errors and exceptions. While the proposed approaches in those studies help software developers collect a list of relevant web pages for an encountered exception, our third study focuses on supporting them with post-search content analysis, more specifically, in finding out the sections of a web page that might help them with the exception in the IDE.

The rest of the chapter is organized as follows–Section 5.2 explains the proposed metrics, algorithms and content extraction technique and Section 5.3 discusses the conducted experiments, results and validation. Section 5.4 identifies the threats to validity, Section 5.5 focuses on the related works, and finally Section 5.6 summarizes the chapter with future works.

5.1 Introduction

While collecting information using traditional web search, developers first use a search engine with a few keywords to get the relevant pages. However, in order to retrieve the required information, they need to go through the pages one by one, which is challenging, and this study focuses on this particular research problem. Both manual checking of a web page for relevant content against an error or an exception and its context (i.e., surroundings, circumference) and working an appropriate solution out are non-trivial tasks, and they require a significant amount of cognitive efforts. The tasks are even more complex and time-consuming with the bulk of irrelevant (i.e., off-topic) and noisy (e.g., advertisement) content in the page. As early as 2005, Gibson et al. [48] estimated that about 40%-50% of web data were *noise*, and they suggested that the ratio would increase due to explosive growth of Internet and advances in information technology. Thus the developers often spend a significant amount of time and effort in searching and then extracting the content of interest from the web pages. Fortunately, automated support in post-search content analysis can greatly benefit them in this regard. For example, extraction and recommendation of relevant sections from a selected web page can help them get rid of information overload and locate the content of interest instantly, which in turn reduce the overall problem-solving efforts by the developers.

A number of existing studies concentrate on *main* (i.e., all noise-free sections) content extraction from a web page by applying different techniques such as template or similar structure detection [39, 58], machine

learning [42, 44, 60], information retrieval, domain and context modeling [46], and page segmentation and filtration [40, 42, 50, 57, 68, 79]. However, no existing studies focus on targeted extraction such as extraction of relevant sections (i.e., sections of interest) from the page. Furthermore, they were not driven by the motivation of developer support, neither did they focus on analyzing any information (e.g., details of an exception) from within the IDE for better extraction of the relevant page contents. Thus those studies fail to direct one to the right (or relevant) sections of the page, and do not help much either in reducing information overload or in locating solution in the page. Furthermore, most of the existing approaches are domain or template specific, and they extract content from the web pages of different domains such as news, consumer products, business and real estates, Wikipedia and social networking. However, none of them deals with programming related web pages, and thus they are also not properly applicable to our research problem.

In this study, we propose a novel web page content recommendation approach that extracts not only all noise-free sections but also relevant sections of a programming related web page by exploiting the technical details of an encountered programming exception in the IDE (i.e., *context-aware* approach). Once a developer searches about an exception using a few keywords, the search engine (e.g., in our case Google) returns a number of pages. Then the real challenge for her is to manually check those pages for relevant content for the encountered exception, and our proposed approach helps her in this regard. The approach analyzes the context (i.e., stack trace and associated code) of an encountered exception in the IDE and the Document Object Model (DOM)[1] tree of a returned web page, and recommends those sections in the page which are relevant to the exception. For example, the code under development (hereby we call it *context code*) in Listing 5.1 triggers an *EOFException*, and the IDE reports the stack trace in Listing 5.2. Our approach analyzes both the content of the returned web page (e.g., Fig. 5.2) and the technical details of the exception (e.g., Listing 5.1 and Listing 5.2), and extracts the relevant section(s) (Fig. 5.1, boxed area from Fig. 5.2) of the page. We integrate Google search API into Eclipse IDE to collect web pages against the developer provided search queries about an exception, and use those pages for the recommendation of relevant sections from them one by one as the developer wishes. In this way, even though Google search may return a lot of web pages for the chosen keywords against the exception, our approach can reduce the burden of the developer by recommending the relevant sections of the pages or even indicating that some particular pages might not have any relevant sections at all for the encountered exception. In the case of the developer interested in a noise-free version of the web page for browsing, the approach discards the noisy sections, and returns the remaining sections of the page. We package our recommendation solution into an Eclipse plug-in prototype, *ContentSuggest*, that can be found online [5].

The proposed approach also complements existing techniques in order to overcome certain limitations. First, the density metrics–*text density* and *link density* proposed by existing literature [57, 79] do not differentiate between regular texts (e.g., news article) and programming related content (e.g., source code segments, stack traces) in a web page, which is essential for effective content recommendation from programming re-

[1] http://en.wikipedia.org/wiki/Document_Object_Model

Listing 5.1: Context code of an EOFException

```
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
ArrayList<Record> currentList = new ArrayList<>();
// restore the number of objects
int size = ois.readInt();
// restore the objects
for (int i = 0; i < size; i++) {
  Record current = (Record) ois.readObject();
  currentList.add(current); }
```

lated web pages. Our approach considers a novel density metric related to such content– *code density* to complement the existing density metrics. Second, the approach introduces a novel idea of leveraging *content relevance* in the extraction and recommendation of web page content. Once noisy sections are discarded, the approach estimates the relevance of different sections of the page against the encountered exception (and its technical details) in the IDE, and recommends the most relevant ones.

We conduct experiments on our approach using a collection of 500 programming related web pages and 150 programming errors and exceptions. We manually develop two gold sets– *main-gold-set* and *relevant-gold-set* by carefully extracting all the noise-free sections and relevant sections of the pages respectively, and use them as *oracles* in order to evaluate our approach. In the case of main (i.e., noise-free) content extraction, our approach extracts content from the web pages with a *precision* of 89.88%, a *recall* of 87.48% and a F_1-*measure* of 87.53%, which are highly promising according to existing relevant literature [42, 57, 79]. In the case of relevant content recommendation, the approach recommends content with a *precision* of 80.50%, a *recall* of 78.39%, and a F_1-*measure* of 76.40%, which are also promising. We compared with four existing approaches in order to validate the performance of our approach in main content extraction, and compared with one existing approach for validation with relevant content recommendation. In the case of main content extraction, our approach outperforms them in terms of *recall* and F_1-*measure* with comparable *precision*, whereas the approach performs significantly better than the existing one in all three performance metrics for relevant content recommendation. Thus the study makes the following contributions:

- We complement the existing metrics, and propose *code density* metric in order to extract and recommend content from programming related web pages.

- We introduce and successfully apply *content relevance* in content extraction from a web page, which in turn provides a mean to support the developers in post-search content analysis and relevant content recommendation.

- We package the proposed solution into an Eclipse plug-in prototype [5], that captures the technical details of an encountered exception in the IDE, and recommends not only all the noise-free content sections but also the relevant sections from a web page resulted from an IDE-based web search.

Listing 5.2: Stack Trace of the Exception in Listing 5.1

```
Exception in thread "main" java.io.EOFException
at java.io.ObjectInputStream$PeekInputStream.readFully(ObjectInputStream.java:2281)
at java.io.ObjectInputStream$BlockDataInputStream.readShort(ObjectInputStream.java
    :2750)
at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:780)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java:280)
at HighScores.<init>(HighScores.java:45)
at HighScores.main(HighScores.java:151)
```

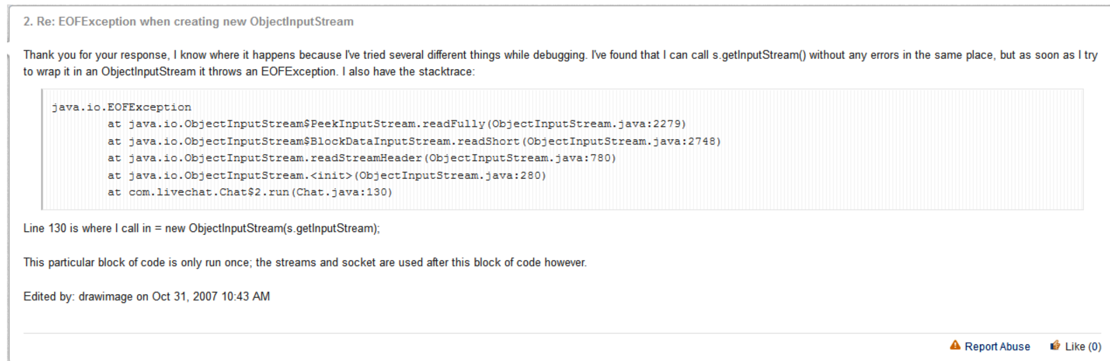

Figure 5.1: An Example Relevant Section for the Exception in Listing 5.2

5.2 Proposed Approach

In this section, we discuss different working modules of our proposed system, proposed metrics and ranking algorithms for different sections in the web page, metric weight estimation, and content extraction technique from the DOM tree of the page.

5.2.1 Working Modules

In Fig. 5.3, the schematic diagram of our proposed approach shows the working modules, and explains different steps required for page content extraction, recommendation and visualization. We package the whole solution as an installable Eclipse plug-in prototype [5], and it has three working modules as follows:

Content Collector: The proposed approach exploits the technical details– stack trace and context code (i.e., a segment of the code that triggers the exception) of an encountered programming exception in order to recommend relevant sections from a selected web page. The *collector module* collects the error message and the stack trace from the active *Console View* (Fig. 5.3-(c)), and the context code from the active text editor (Fig. 5.3-(d)) in the IDE. It also collects the HTML source of a selected web page (Fig. 5.3-(a)). Once a developer selects a result page in the list and requests for relevant or main (i.e., noise-free) content (e.g., Fig. 5.3-(b)), the *collector module* downloads the HTML source of the page, and sends the HTML source as well as the exception details to the *extractor module*.

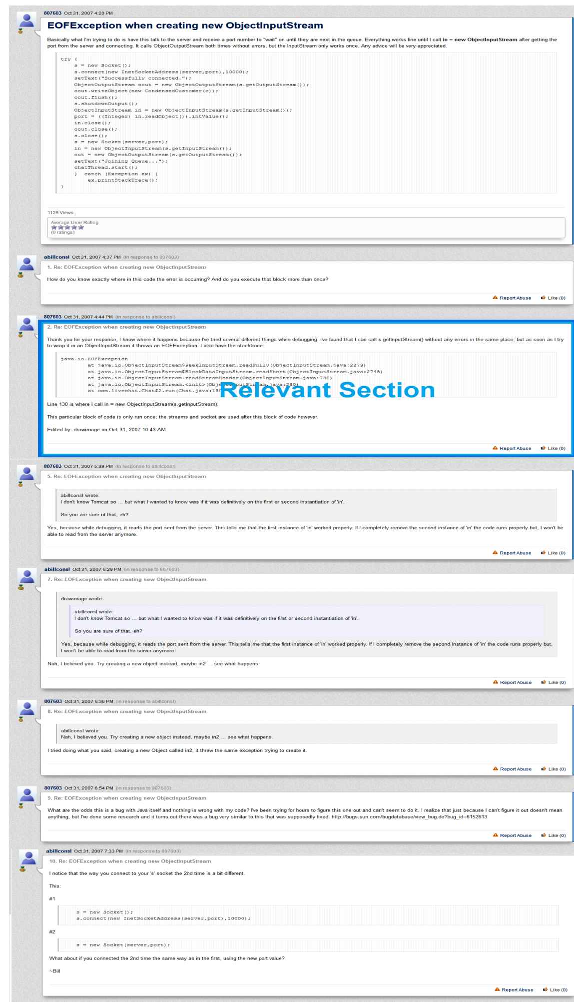

Figure 5.2: Relevant Section(s) in the Webpage

Content Extractor: The *extractor module* (i.e., dashed rectangle, Fig. 5.3-(e)) parses each of the tags of a HTML page, and develops a DOM tree. It then analyzes each of the nodes in that tree, calculates its *content density* and *content relevance* (Section 5.2.2), and assigns a *content score*. The module then discards the noisy nodes based on their calculated scores and existing heuristics [79], and returns main (i.e., noise-free sections) content of the page. It also identifies the DOM tree nodes that are the most relevant to the encountered exception in the IDE, and recommends the corresponding sections in the page as relevant content. It should be noted that the *extractor module* returns each type of content as HTML source so that the organization of texts and other elements (i.e., layout of the page) is preserved during visualization.

Content Visualizer: The *visualizer module* consists of two panels– one visualizes the relevant sections and the other visualizes the actual or noise-free version of a web page. We use SWT browser widget in order to display the HTML content within the IDE. The relevant content panel (Fig. 5.3-(f)) displays only the most relevant sections of a page recommended by the *extractor module*. It also highlights different program elements of interest– stack traces and code segments in those sections. The idea is to help a developer either

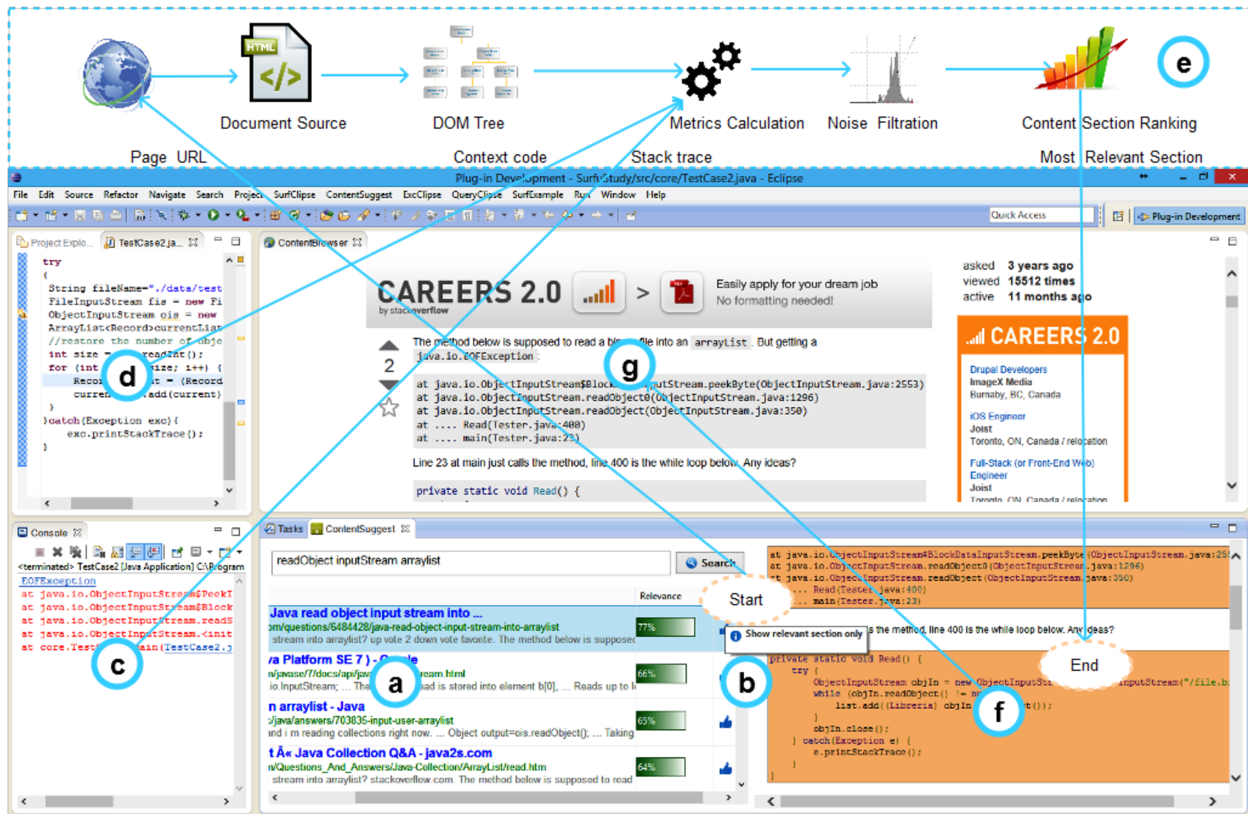

Figure 5.3: Schematic Diagram of Proposed Approach (ContentSuggest)

locate the content of interest instantly or decide if the selected page is worth browsing or not. Once she is convinced with the relevant sections, she can browse the noise-free version or the actual page using the other panel (e.g., Fig. 5.3-(g)) for further analysis. It should be noted that the result panel (Fig. 5.3-(a, b)) visualizes the estimated relevance of each result page against the exception in the IDE by analyzing the meta description of the page collected from the search engine. This visualization helps the developer choose an appropriate result page in the first place during searching for a solution.

5.2.2 Proposed Metrics

In this section, we discuss our proposed metrics that are used to extract and recommend both relevant section(s) and all noise-free sections of a web page. The metrics analyze not only the legitimacy of a content section in the page but also the relevance of the section against an exception in the IDE.

Content Density (CTD)

Existing studies [57, 79] propose and apply two density metrics–*text* or *word density*, and *link density* for *noise-free* content extraction from a web page. However, these metrics are based on regular texts (e.g., news article), and they are neither properly applicable nor sufficient enough for content extraction from programming related web pages. These pages contain items beyond regular texts such as stack traces and

code segments. We thus modify existing metrics, introduce a new density metric, and finally propose a composite density metric.

Text Density (TD): *Text Density* represents the amount of any textual content each of the HTML tags in the web page contains on average [79]. Thus in the DOM tree, *text density* (TD_i) of a node is calculated considering its number of child nodes (T_i) (i.e., inner nodes) and the amount of texts (C_i) it contains in the leaf nodes as follows:

$$TD_i = \frac{C_i}{T_i} \tag{5.1}$$

Link Density (LD): *Link Density* represents the amount of linked (i.e., noisy) texts each of the HTML tags contains on average. Existing literature [57, 79] considers any linked text in the web page as *noise*; however, in our research, we make a careful choice about them. We analyze the relevance of each linked text element against the exception of interest, and consider the element as *noise* only if its relevance is below a heuristic threshold (η=0.75). We otherwise consider it as a legitimate textual element. Thus in the DOM tree, the *link density* (LD_i) of a node i is calculated considering its number of child nodes (T_i) (i.e., inner nodes) and the amount of linked or noisy texts (LC_i) it contains in the leaf nodes as follows:

$$LD_i = \frac{LC_i}{T_i} \tag{5.2}$$

We consider each *<a>* tag, and check its relevance before considering it as *noise*. As Sun et al. [79] suggest, we also consider *<input>* and *<button>* as linked elements, and their content as linked texts.

Code Density (CD): *Code Density* represents the amount of *code related texts* each of the HTML tags contains on average. Programming related web pages generally contain different program elements such as *stack traces* and *code segments*, and they are of significant interest to the developers. The developers often analyze or reuse them in order to solve their programming problems. We believe that the code related elements complement the discussion texts about programming, and thus *code density* can be considered as an important indicator of legitimacy of a programming related web page. In the DOM tree, the code density (CD_i) of a node i is calculated considering its number of child nodes (T_i) (i.e., inner nodes) and the amount of code related texts (CC_i) it contains in the leaf nodes as follows:

$$CD_i = \frac{CC_i}{T_i} \tag{5.3}$$

We observe that code related elements are generally posted in the page using *<code>*,*<pre>* and *<blockquote>* HTML tags, and we consider their texts as *code related texts* in density calculation.

While *text density* metric represents a generalized form of density for all kinds of text, *code density* and *link density* point to special types of text. *Code density* can be considered as a heuristic measure of programming elements in the text, whereas *link density* is a similar measure for *noise* in the content. In our research, we consider all three metrics of an HTML tag i, and propose a *log-based composite density metric* called *content density* (CTD_i), which is partially motivated by the idea of Sun et al. [79].

$$CTD_i = (TD_i + \frac{CD_i}{TD_i}) \times log_{ln(\frac{TD_i \times LD_i}{\neg LD_i} + \frac{LD_b \times TD_i}{TD_b} + e)}(\frac{TD_i}{LD_i} + \frac{CD_i}{TD_i}) \quad (5.4)$$

Here, TD_i, CD_i, LD_i and $\neg LD_i$ represent *text density*, *code density*, *link density* and *non-link density* of the HTML tag i respectively. TD_b and LD_b represent the *text density* and *link density* of *body* tag respectively. In Equation (5.4), $\frac{TD_i}{LD_i}$ is a measure of the proportion of linked texts. When a tag has higher *link density*, $\frac{LD_i}{\neg LD_i} \times TD_i$ expression increases the log base, $\frac{TD_i}{LD_i}$ gets a lower value, and thus overall *content density* is penalized. However, $\frac{LD_b \times TD_i}{TD_b}$ expression maintains the balance between these two interacting parts, and prevents a lengthy and homogeneous text block from getting a higher value or a single line text (e.g., page title) from getting a low value. Moreover, we introduce the programming text proportion of a tag, $\frac{CD_i}{TD_i}$, which improves the overall *content density* metric for the HTML tag that contains both programming related texts and comprehensive regular texts.

Content Relevance (CTR)

Existing studies [57, 79] apply different density metrics in order to discard the noisy sections (e.g., advertisements, navigation menus) and extract the legitimate sections from a web page. However, these metrics are not sufficient enough for relevant content extraction from the page (i.e., our research problem). We thus leverage the technical details of the encountered exception in the IDE, and propose three relevance metrics in order to determine the relevance of different sections in the web page against the exception.

Text Relevance (TR): *Text relevance* estimates the relevance of the textual content within an HTML tag against the exception (and its context) of interest (i.e., in the IDE). The *context of the exception* is represented as a list of keywords extracted from corresponding stack trace and context code (Section 5.3.3). For example Listing 5.3 shows the context of our showcase exception in Listing 5.1 and Listing 5.2. We calculate *cosine similarity* between that keyword list and the texts within each tag in the page. Since *cosine similarity measure* represents the lexical similarity between two texts, we consider the calculated measure as an estimate of lexical relevance for the tag against the exception. The relevance estimate values from zero to one, where one refers to complete lexical relevance and vice versa.

Code Relevance (CR): *Code relevance* is an estimate of average relevance of a code segment (e.g., context code) or a stack trace block embedded within an HTML tag against that of the exception of interest. In the DOM tree of an HTML page, in order to estimate *code relevance* of a node, we analyze three types of tags– *<code>*,*<pre>* and *<blockquote>* under the node. According to traditional heuristics [72], these tags generally contain the program elements, and we apply two different techniques for stack traces and code segments respectively in order to estimate their relevance.

Stack trace of a programming exception contains an error message followed by a list of method call references that point to possible error locations in the code. As suggested by Rahman et al. [72], we develop separate token list by extracting suitable tokens (e.g., class name, method name) from each of the stack

Listing 5.3: Used Context of the Exception

```
Exception in thread "main" java.io.EOFException readInt ObjectInputStream
init readStreamHeader BlockDataInputStream  readObject readShort
add main readFully FileInputStream Record ArrayList PeekInputStream
```

trace within the HTML tag and the stack trace in the IDE respectively. We then calculate *cosine similarity* between the two lists, and consider the measure as an estimate of relevance for the stack trace within the HTML tag against the exception of interest.

In order to determine the relevance of a code segment within the HTML tag against an exception of interest, we collect the context code of the exception, and apply a state-of-the-art code clone detection technique by Roy and Cordy [75]. The technique determines the *longest common subsequence* of source code tokens (S_{lcs}) between two code segments. We use it to determine the code similarity of the code segment in the HTML tag against the context code as follows, where S_{total} refers to the set of all tokens collected from the context code of the target exception.

$$S_{ccx} = \frac{|S_{lcs}|}{|S_{total}|} \tag{5.5}$$

Once the relevance of all the program elements (i.e., stack traces and code segments) under an HTML tag are estimated, we average the estimates, and consider it as the *code relevance* (CR_i) for the tag i as follows, where N is number of program elements found under the tag in the DOM tree.

$$CR_i = \frac{1}{N}\sum_{j=1}^{N} CR_j \tag{5.6}$$

While *text relevance* focuses on the relevance of any textual element within an HTML tag, *code relevance* estimates the relevance of program elements within it. We combine both relevance metrics in order to determine the *composite relevance metric* called *content relevance* (CTR) as follows:

$$CTR_i = \alpha \times TR_i + \beta \times CR_i \tag{5.7}$$

Here α and β are the relative weights (i.e., importance) of the corresponding relevance metrics, which are estimated using a machine learning based approach (Section 5.2.4).

5.2.3 Content Score Calculation

We consider two aspects–*density* and *relevance* of different content sections in the web page in order to extract all the noise-free ones and more importantly, the relevant ones. While the density metrics focus on the legitimacy (i.e., purity) of the content in the page, relevance metrics check the relevance of the same content against the programming problem (i.e., an error or an exception) at hand. The idea is to recommend both legitimate (i.e., noise-free) and relevant content to the developers. We thus combine both aspects and propose a *composite score metric* called *content score* (CTS_i) for each of the HTML tags in the web page as follows. It should be mentioned that we use normalized metrics in *content score* calculation for relevant

content extraction in order to mitigate the bias of any of the two aspects considered.

$$CTS_i = \gamma \times CTD_i + \delta \times CTR_i \tag{5.8}$$

Here γ and δ are the relative weights (i.e., importance) of the corresponding density and relevance metrics, which are estimated using a machine learning based approach involving logistic regression (Section 5.2.4).

5.2.4 Metric Weight Estimation

In order to determine the relative weights of two relevance metrics–*text relevance* and *code relevance* and two composite metrics–*content density* and *content relevance*, we choose 50 random web pages from the dataset, and collect the corresponding metrics of 705 text blocks (i.e., page sections) by our approach from those pages. It should be noted that the individual relevance metrics are treated equally in terms of relative importance in the calculation of *content relevance* metric at this stage. We also identify whether each of those blocks is included in the gold content or not, which provides a *binary class label* against the set of features (i.e., metrics) for the text block. We then feed the feature (i.e., metric) values and class labels of the 705 block samples to *Weka* tool [29] that returns a logistic regression based classifier model [17] which is validated with ten-fold cross validation. 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., text block) 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 metrics [61]. For the sake of simplicity and in order to reduce bias, we normalize those coefficients, and consider a heuristic weight α=1.00 for *text relevance* and β=0.59 for *code relevance* metrics. In case of composite density and relevance metrics, we find that the proposed approach performs significantly well with equal relative weights assigned. Thus we consider a heuristic weight of 1.00 for both the composite metrics.

5.2.5 Document Content Extraction

Once *content score* of each of the HTML tags in a page is calculated, the noisy elements are discarded based on a *heuristic threshold*. Since <*body*> tag contains the whole content and encloses all other tags in the page, its *content score* sums up average density and average relevance estimates of the whole page. As Sun et al. [79] suggest in case of *text density*, in our research, we consider the *content score* of <*body*> tag as the *threshold score*, and we exploit the DOM tree of the HTML page in order to discard the noisy elements. An HTML page is generally divided into a set of identifiable blocks which are represented as the child nodes of *body* node in the DOM tree. We check each of those child nodes for its *content score*, and discard the ones having scores less than the threshold. We then explore each of the remaining child nodes, and find out its *inner node* with the highest score [79]. The highest score of a node indicates that the corresponding tag in the HTML page contains the most legitimate content in terms of different density and relevance estimates. In order to extract the main (i.e., noise-free) content, we keep the highest scored node along with its child nodes, and mark them as *content node*. We apply the same process recursively for each node in the DOM

tree, and finally we get each node in the tree annotated as either *content* or *noise*. We then discard the noisy nodes, and extract the HTML tags corresponding to the remaining nodes in the tree as the main (i.e., noise-free) content of the page. In case of relevant content extraction, we analyze the relevance of each of the child nodes under *body* node in the DOM tree, and choose the *highest scored relevant node*. The idea is to ensure that the corresponding recommended sections in the page are not only relevant but also rich in terms of legitimacy (i.e., noise-free).

For example, our ranking algorithm returns these metric values– TD=32.74, LD=2.88, CD=24.29, CTD=0.02, CR=0.84, TR=0.83, CTR=0.99, and CTS=1.0144, for the page section in Fig. 5.1. The section outperforms other sections in Fig. 5.2 both in terms of legitimacy (i.e., density) and relevance with the target stack trace in Listing 5.2. Thus our approach marks the section as a relevant one, and extracts it for recommendation.

5.3 Experimental Design, Results and Validation

5.3.1 Dataset Preparation

In our experiments, we use a dataset of 500 web pages (hereby *main set*) related to 150 programming exceptions [4], a subset of *main set* containing 250 web pages (hereby *relevant set*) related to 80 exceptions, and technical details (e.g., error messages, stack traces and context code) of the exceptions. Most of the technical details and the web pages are collected from our previous work– *SurfClipse* [72], and a few of them are collected from different online sources such as Pastebin [20] and StackOverflow. We develop two gold sets– *main-gold-set* and *relevant-gold-set* [4] for *main set* and *relevant set* respectively, and use them to evaluate our approach. The first gold set contains all noise-free sections of a web page whereas the other contains only the relevant sections of the page. Both gold sets are developed through extensive manual analysis, and we spent about 40-50 working hours. One of the authors carefully extracted the desired sections from each of the pages, which were also validated by the peer. We consider different items such as advertisements, navigation menus, copyright notice, repeated discussion texts, and insignificant comments as *noisy* content, and discard them to obtain the main (i.e., noise-free) content of a page [79]. In case of relevant content extraction, we analyze the relevance of each of the noise-free sections in the page against the exception of interest. We note that most of the pages in the dataset generally start with a section that describes a technical problem involving a programming error or exception. The section contains a question title and different program elements such as stack traces and code segments. We look for such a section in each page in the dataset, and extract it as the most relevant section for the page. Given that determining relevance of a page section is a subjective approach, and in order to reduce the bias, we look for relevant stack traces or code segments in a candidate section and then extract it applying our best judgement.

Table 5.1: Dataset Statistics

Dataset	SO Pages (DSO)	Non-SO Pages (¬DSO)	All Pages (D)	Exceptions
Main set	208 (41.60%)	292 (58.40%)	500 (100%)	150
Relevant set	101 (40.40%)	149 (59.60%)	250 (100%)	80

SO=StackOverflow, **DSO**=Data from StackOverflow

Both *main set* and *relevant set* contain about 40% of the pages from StackOverflow Q & A site, and we were interested to contrast them with other pages. We break each set into two more subsets–*StackOverflow pages (DSO)* and *Non-StackOverflow pages (¬DSO)*, and thus, we use three sets–*DSO,* *¬DSO* and *D* for each of *main* and *relevant content* extraction, where *D* combines both *DSO* and *¬DSO* sets. Table 5.1 shows the statistics of the different sets of web pages.

5.3.2 Traditional Web Page Content Recommendation

We investigate existing supports for page content recommendation by traditional search engines such as Google, Bing and Yahoo. Each of the search providers returns results in the ranked list displaying the title and a minimal description for each of the result pages. They also highlight the terms in the title and the description that match directly with the terms in search query. However, the limited overview (i.e., description) and information highlighting are not often enough, and a developer is forced to browse the result page in order to determine its relevance against an exception of interest and its context. During the browsing, one can exploit the *search feature* provided by the browsers; however, still it is not much helpful to find out the most relevant or interesting sections from a web page.

5.3.3 Exception Context Representation

In our research, we apply not only the density metrics but also the relevance estimates for main (i.e., noise-free) and relevant content extraction. Each page in the dataset is relevant to a particular exception, and we exploit the technical details (e.g., stack trace and context code) of the corresponding exception for relevance estimation of different sections in the page. We analyze the stack trace (e.g., Listing 5.2) and extract different tokens such as *package name*, *class name* and *method name* from each of the method call references. We also analyze the context code (e.g., Listing 5.1) of the exception and collect *class name* and *method name* tokens. We use *Javaparser*[2] for compilable code and an *island parser* for non-compilable code in order to extract the tokens [72]. We then combine tokens both from the stack trace and the context code, and append the exception name along with the exception message (i.e., highlighted in Listing 5.2) to the combined set. Thus we get a simplified context for the exception of interest, which is used in *text relevance* estimation (Section 5.2.2). For example, Listing 5.3 shows the context for an *EOFException* with stack trace in Listing 5.2 and context code in Listing 5.1.

[2]http://code.google.com/p/javaparser/

5.3.4 Performance Metrics

Our proposed approach is aligned with the research areas of information retrieval and recommendation systems, and we thus use a list of performance metrics from those areas in order to evaluate our approach as follows [51, 72, 79]:

Mean Precision (MP): *Precision* determines the percentage of retrieved content that is expected (i.e., main content, relevant content) from a web page. In our research, we compare the retrieved content by our approach with the manually prepared gold sets. As Sun et al. [79] suggest, we use *longest common subsequence* of tokens between retrieved content and gold content. Thus *precision* can be determined as follows, where a refers to the token sequence of retrieved main or relevant content and b refers to that of the corresponding gold content.

$$P = \frac{|LCS(a,b)|}{|a|}, \quad MP = \frac{\sum_{i=1}^{N} P_i}{N} \tag{5.9}$$

Mean Precision (MP) averages the *precision* measures for all the web pages (N) in the dataset.

Mean Recall (MR): *Recall* measure determines the percentage of the expected content (i.e., main content, relevant content) of a web page that is retrieved by a system. We calculate the *recall* of a system as follows:

$$R = \frac{|LCS(a,b)|}{|b|}, \quad MR = \frac{\sum_{i=1}^{N} R_i}{N} \tag{5.10}$$

Mean Recall (MR) averages the *recall* measures for all the pages (N) in the dataset.

Mean F_1-measure (MF): While each of *precision* and *recall* focuses on a particular aspect of the performance of a system, F_1-measure is a combined and more meaningful metric for evaluation[3]. We calculate F_1-measure from the harmonic mean of *precision* and *recall* as follows [79]:

$$F_1 = \frac{2 \times P \times R}{P + R}, \quad MF = \frac{\sum_{i=1}^{N} F_{1i}}{N} \tag{5.11}$$

Mean F_1 (MF) averages all such measures.

5.3.5 Experimental Results

We conduct experiments with two datasets– *main set* and *relevant set*, and extract main (i.e., noise-free) content and relevant content respectively from their pages. We then check those extracted content against the carefully prepared *main-gold-set* and *relevant-gold-set* respectively, and evaluate the performance of our approach. Table 5.2 and Table 5.3 summarize the findings of our evaluation.

[3] http://stats.stackexchange.com/questions/49226/

Table 5.3: Experimental Results for Different Aspects of Page Content

Content Type	Score Combination	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All Pages (D)
Main Content	{Content Density (CTD)}	MP	90.89%	88.86%	89.71%
		MR	89.38%	86.20%	87.53%
		MF	89.85%	85.75%	**87.45%**
	{Content Relevance (CTR)}	MP	89.80%	75.40%	81.39%
		MR	25.66%	37.83%	32.76%
		MF	33.82%	45.31%	**40.53%**
	{Density (CTD), Relevance (CTR)}	MP	91.27%	88.90%	89.88%
		MR	89.27%	86.20%	87.48%
		MF	90.00%	85.76%	**87.53%**
Relevant Content	{Content Density (CTD)}	MP	**50.91%**	49.50%	50.07%
		MR	91.74%	75.71%	82.18%
		MF	**62.32%**	53.76%	**57.22%**
	{Content Relevance (CTR)}	MP	**86.63%**	69.17%	76.23%
		MR	52.17%	57.66%	55.44%
		MF	61.07%	55.88%	**57.98%**
	{Density (CTD), Relevance (CTR)}	MP	89.91%	74.12%	80.50%
		MR	74.90%	80.76%	78.39%
		MF	80.07%	73.91%	**76.40%**

Table 5.2: Results of Experiments on Main (i.e., noise-free) and Relevant Content

Content Type	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All Pages (D)
Main content	MP	91.27%	88.90%	**89.88%**
	MR	89.27%	86.20%	**87.48%**
	MF	90.00%	85.76%	**87.53%**
Relevant content	MP	**89.91%**	**74.12%**	**80.50%**
	MR	74.90%	80.76%	78.39%
	MF	80.07%	73.91%	**76.40%**

From Table 5.2, we note that our proposed approach extracts main content with a *mean precision* of 89.88% and a *mean recall* of 87.48%, which are highly promising. In the case of main content extraction from a web page, both *precision* and *recall* are important, and our approach also performs well in terms of the combined metric, F_1*-measure* (e.g., 87.53%). The *main set* contains about 41.60% of the pages from StackOverflow. During gold set preparation, we notice that the pages from StackOverflow (hereby SO) follow a consistent structure with relatively less noise, and relevant content sections are more precise and persistent than those in the pages from other web sites, which are helpful for content extraction. We were interested to check the performance of our approach against three different subsets of *main set–SO Pages*, *Non-SO Pages* and *All Pages* (Table 5.1). In Table 5.2, we note that the approach performs almost equally well for all the subsets with different types (i.e., websites) and different sizes (i.e., number of pages), which demonstrates the robustness and generality of our approach.

In case of relevant content, our approach extracts content with a *mean precision* of 80.50%, a *mean recall* of 78.39%, and a *mean F_1-measure* of 76.40%, which are also promising. We use the *relevant set* (Table 5.1) for the experiments, which contains about 40.40% of the pages are from StackOverflow. We conduct experiments with different subsets of pages, and find that the approach provides the most *precise recommendation* of 89.91% with StackOverflow pages. The aforementioned scenario of StackOverflow might partially help our approach to perform better; however, the approach also recommends with a *mean precision* of 74.12% with non-StackOverflow web pages, which is promising and significantly better than that of the existing approaches (Table 5.4).

Table 5.3 investigates the effectiveness of the two aspects–*content density* and *content relevance* that we propose for content extraction from a web page. We consider each of those aspects in isolation as well as in combination, and evaluate our approach with different sets of web pages for different types of extraction–*main content* and *relevant content*. In case of *content density*, the proposed approach performs significantly well in terms of all three performance metrics for main content extraction with all three subsets–*StackOverflow pages (DSO), Non-StackOverflow pages (¬DSO)* and *D* of *main set*. However, the metric is found not much effective in case of relevant content extraction with any of the subsets of *relevant set*, and the approach provides imprecise results. For example, it extracts the relevant content from a web page of any of the three subsets (*DSO, ¬DSO* and *D*) with a maximum *mean precision* of 50.91% and a *mean F_1-measure* of 62.32%. In case of *content relevance* metric, the proposed approach extracts relevant content from a web page with relatively better precision (e.g., 86.63%); however, the recall rates are poor both in main content and relevant content extraction. On the other hand, when we combine both the density and relevance metrics, we experience significant improvement in all three performance metrics for both types (e.g., main and relevant) of extraction with each of the sets of web pages. For example, main (i.e., noise-free) and relevant sections of a page are extracted by our approach with a mean *F_1-measure* of 87.53% and 76.40% respectively.

It should be noted that in case of main (i.e., noise-free) content extraction, much improvements in performance are not achieved with the combination of metrics compared to with density metric only. The finding disproves our primary intuition about the effectiveness of relevance metric in main content extraction. However, the finding from the relevant content extraction clearly shows the effectiveness of the combination of density and relevance metrics which is one of our primary objectives of this work.

5.3.6 Comparison with Existing approaches

In order to validate the performance of our approach, we compare with four existing approaches [50, 51, 68, 79] for main content extraction, and one existing approach [79] for relevant content extraction from the literature. It should be noted that those four approaches only extract main content from web pages, and the rest one is adapted for relevant content extraction, whereas our approach extracts both main (i.e., noise-free) content and relevant content from the pages. Table 5.4 summarizes the findings from our comparative study.

Table 5.4: Comparison with Existing Approaches

Content Type	Content Extractor	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All Pages (D)
Main content	Sun et al. [79]	MP	**80.61%**	**78.70%**	79.49%
		MR	86.41%	**75.67%**	80.14%
		MF	83.14%	**75.48%**	**78.67%**
	ACCB [50]	MP	90.65%	**93.07%**	92.06%
		MR	77.32%	**79.98%**	**78.87%**
		MF	83.07%	84.64%	**83.99%**
	DSC [68]	MP	98.27%	91.05%	**94.05%**
		MR	62.69%	67.56%	**65.53%**
		MF	74.54%	75.26%	**74.96%**
	TCCB [51]	MP	**96.47%**	88.89%	92.04%
		MR	**65.98%**	61.43%	63.32%
		MF	76.96%	68.70%	**72.14%**
	Proposed Approach (Density Only)	MP	90.89%	88.86%	89.71%
		MR	89.38%	86.20%	87.53%
		MF	89.85%	85.75%	**87.45%**
	Proposed Approach (Density and Relevance)	MP	**91.27%**	**88.90%**	**89.88%**
		MR	**89.27%**	**86.20%**	**87.48%**
		MF	90.00%	**85.76%**	**87.53%**
Relevant content	Sun et al. [79]	MP	52.63%	38.89%	**44.44%**
		MR	86.49%	41.84%	**59.88%**
		MF	62.57%	34.49%	**45.84%**
	Proposed Approach (Density and Relevance)	MP	89.91%	74.12%	**80.50%**
		MR	74.90%	80.76%	**78.39%**
		MF	80.07%	73.91%	**76.40%**

Sun et al. [79] propose a main (i.e., noise-free) content extraction approach, where they exploit the DOM-tree of a web page and apply a text density-based scoring technique. We collect an implementation[4] of the approach from corresponding authors, and conduct experiments with the approach using our dataset (Table 5.1). From Table 5.4, we note that our approach performs significantly better than their approach in terms of all three performance metrics with each of the sets (e.g., *StackOverflow Pages (DSO), Non-StackOverflow Pages (¬DSO)* and *All Pages (D)*) of web pages. For example, with *Non-StackOverflow Pages*, the approach by Sun et al. can extract main content with a *mean precision* of 78.70%, a *mean recall* of 75.67% and a *mean F_1-measure* of 75.48%. On the other hand, our approach extracts such content from the same dataset with 88.90% *precision*, 86.20% *recall and* 85.76% *F_1-measure*. Their approach performs well only with StackOverflow pages, whereas our approach performs more consistently with different sets of web pages, and also provides relatively more promising results. One can think of the *Density Only* version of our approach

[4] https://github.com/FeiSun/ContentExtraction

equivalent to the approach of Sun et al. from theoretical perspective; however, experimental results (Table 5.4) show that our density metric is more effective than theirs in extracting main content from a web page.

Pinto et al. [68] propose another content extraction system that exploits *Document Slope Curves (DSC)* filtration technique and locates the regions of a web page where word tokens are more frequent than HTML tag tokens using a windowing technique. We collect the author's implementation of the approach [9], and compare with our approach. From Table 5.4, we find that their approach provides results with relatively higher precisions, but the recall rates are quite poor compared to those of our approach. For example, the approach extracts main content from our dataset with a *mean precision* of 94.05%, but returns only 65.53% (i.e., *recall*) of the expected page content compared to 87.48% (i.e., *recall*) of our approach.

Gottron [50] introduces a content extraction algorithm that identifies homogeneously formatted texts from the HTML source token sequence of a web page. We collect the author's implementation [9] and compare with two different versions of the algorithm–*ACCB* [50] and *TCCB* [50, 51]. From Table 5.4, we note that both approaches provide relatively more precise results compared to our approach; however, their *recall rates* are relatively poor. For example, ACCB and TCCB returns 79.98% (i.e., *recall*) and 65.98% (i.e., *recall*) of the page content respectively at a maximum level from any of the sets of pages, and they extract main content with 93.07% and 96.47% *precisions* respectively. On other hand, our approach returns about 87.48% (i.e., *recall*) page content with 89.88% *precision*, which demonstrate the robustness of our approach. More interestingly, it extracts 89.27% (i.e., *recall*) of the page content with 91.27% *precision* for StackOverflow pages. It should be noted that ACCB is relatively closer to our approach in terms of F_1-*measure* (e.g., about 84%); however, it performs comparatively poor in terms of *recall* which is 78.87%.

We also validate the performance of our approach for relevant content extraction, where we adapt and implement an existing approach of Sun et al. [79] in our working environment for comparison. To our knowledge, no existing approaches focus on such content extraction or recommendation despite of its appealing motivation. We find the approach of Sun et al. to be the most aligned for such purpose and it also applies metrics similar to ours for main (i.e., noise-free) content extraction. We choose the highest scored section in terms of *text density* [79] as the most relevant section of a web page. However, from Table 5.4, we note that the density-based metric of the existing approach fails to identify and extract the relevant sections of a web page satisfactorily. For example, their approach recommends relevant content with a *mean precision* of 44.44% and a *mean recall* of 59.88%, which provide a *mean* F_1-*measure* of 45.84%. On the other hand, our proposed approach recommends such content with a *precision* of 80.50%, a *recall* of 78.39% and a F_1-*measure* of 76.40%. The other approaches [50, 51, 68] are not meant for relevant content extraction; however, the way they extract different content sections from a web page, they are less likely to identify and extract the relevant content sections from a web page.

5.4 Threats to Validity

In our research, we note a few issues worthy of discussion. First, we remove all *<style>* and *<script>* tags from an HTML page during parsing, which are often responsible for dynamic loading of noisy sections such as advertisement banners or irrelevant widgets. Thus the look and feel of the pages gets changed, which is a potential threat to our work. In order to mitigate that threat, we add a customized style where we highlight different artifacts of interest (e.g., stack traces, code segments, class or method name) and visualize the relevance of each of the retrieved sections in the page with the exception of interest.

Second, our approach provides a heuristic estimate of relevance for each of the result pages in the ranked list from the search engine (Fig. 5.3-(b)). One might suggest that using similar heuristics the search results could be ranked against the context of the given error or exception. While this has potential, we note that the search engine (e.g., Google) provides a satisfactory ranking with the error message of an exception as the search query for most of the time. Furthermore, in this research, our objective is to complement the existing ranking with relevant information from the web page, and to help the developers in informed decision-making with browsing during post-search content analysis.

Third, the lack of a fully-fledged user study that evaluates the usability of our approach is a potential threat. However, our objective was to focus on the technical aspects of the approach. Furthermore, in order at least partially evaluate the usability, we conduct a limited user study with five participants (i.e., graduate research students), where three of them have professional software development experience. We ask them six questions about the customized style of the page, relevance visualization of each page in the search result and different page sections, highlighting of the artifacts of interest, and the IDE-based information search. Five out of five participants agree that the proposed approach is likely to be more helpful than traditional browsers in problem solving for the developers. All of them feel comfortable with the modified layout and style, and they even speculate that the relevance visualization feature might be really helpful in extracting the desired information from the web page. The details of this mini user study could be found online [4]. However, a fully-fledged user study is required to explore the true potential of our approach which is more appropriate for future work.

5.5 Related Work

A number of existing studies are conducted on web page content extraction, and they apply different techniques such as template or similar structure detection [39, 58], machine learning [42, 44, 60], information retrieval, domain modeling [46], and page segmentation and filtration [40, 42, 50, 57, 68, 79]. The last group of techniques–*page segmentation and noise filtration* are closely related to our research in terms of working methodologies and goals. Sun et al. [79] exploit link elements for the filtration of noisy sections of a web page. The approach by Pinto et al. [68] is actually designed with a table layout-based architecture of the web page

in mind, which may not be applicable for modern complex websites. The two versions– TCCB and ACCB of the approach by Gottron [50] based on *Code Content Blurring (CCB)* are only tested with news websites containing simple and regular structures (i.e., homogeneous text blocks). We compare with all four above approaches, and find that our approach performs relatively better than all of them in terms of *recall* and F_1*-measure*, and also provides a comparable *precision*. For a detailed comparison, the readers are referred to Section 5.3.6.

The other studies use different methodologies and are not closely related to our work, and we do not compare with them in our experiments. Furche et al. [46] analyze real state websites and extract property and price related information. They exploit a domain-specific model for content extraction which may not be applicable for programming related websites. Chun et al. [42] analyze news-based websites, extract different densitometric features, and apply a machine learning classifier (C4.5) in order to classify legitimate and noisy content sections. Their approach is subject to the amount or quality of the training data as well as the performance of the classifier. Cafarella [39] focuses on Wikipedia pages, identifies the tabular structures, and mines different factual information (e.g., list of American presidents) from the pages. Thus, while other approaches focus on extracting the noise-free sections or mining the factual or commercial data from the news-based, real state or Wikipedia pages, our approach attempts to support software developers in finding context-relevant information from the programming related pages. From technical point of view, it proposes and leverages a novel metric–*content relevance* for content (e.g., relevant) extraction, which was not considered by any of the existing approaches.

5.6 Summary

While our previous two studies focus on recommending useful search queries (Chapter 4) and relevant web pages (Chapter 3) for programming errors and exceptions, in this chapter, we propose a novel web page content recommendation approach that extracts and recommends not only all the noise-free sections but also the sections of a web page those are relevant to an encountered exception in the IDE. We also propose and successfully apply a novel metric, *content relevance*, in the extraction of relevant sections from the page. Extensive studies with 500 web pages related to 150 programming errors and exceptions show that our approach extracts main content with a F_1*-measure* of 87.53%, and relevant content with a F_1*-measure* of 76.40%, which are promising. We compared with four existing approaches in case of main content and one approach in the case of relevant content extraction, and found that our approach outperformed them in terms of *recall* and F_1*-measure* for the first case, and in terms of all three metrics for the second case. In order to validate the applicability of our proposed approach in real life problem solving, a fully-fledged user study should be conducted, and we did that in Chapter 7. While the first three studies (Chapter 3, Chapter 4 and Chapter 5) provide automated supports in the resolution of programming errors and exceptions, studies show that developers often use the exception handling features ineffectively and thus they also need

automated support in handling the exceptions effectively. In the next chapter (Chapter 6), we thus propose a recommendation approach that recommends suitable code examples for exception handling from different online repositories by analyzing the code under development in the IDE.

Chapter 6

IDE-Based Context-Aware Exception Handling Code Example Recommendation

Software developers often either misuse the exception handling features or they use them ineffectively in real life software development. They even consider the effective exception handling as a daunting task, and one way to support them in this regard is to recommend readily available relevant code examples those contain high quality handlers of the exceptions of their interest. While the previous studies (Chapter 3, Chapter 4 and Chapter 5) are related with web search, our fourth study in this chapter focuses on a *context-aware* recommendation system that recommends relevant code examples for exception handling.

The rest of the chapter is organized as follows– Section 6.2 presents a motivating example and Section 6.3 explains the proposed metrics, weight estimation technique and the ranking algorithms. Section 6.4 discusses the conducted experiments, results and case study, Section 6.5 identifies the threats to validity, Section 6.6 focuses on the related works, and finally Section 6.7 summarizes the chapter with future works.

6.1 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 [38] or misuse them in the real life software development [77]. Cabral and Marques [38] 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 handlers is a daunting task. One way to benefit both the developer 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 [41, 49, 74] or comparative field studies [38, 47], visualization [77], and recommendation of code examples [33]. Barbosa et al. [33] 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 study, 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* [7], 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 in the IDE and the quality of the exception handlers in those code examples.

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 [33, 55]. 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 [75] 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. The integration makes the corpus for recommendation dynamic, constantly evolving, and synchronized with a number of mature and popular open source projects hosted online.

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 [20]. 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 pre-*

cision, 76.70% *recall* and 86.15% *recommendation accuracy*. These results are found promising according to the existing relevant studies [59, 72, 81]. We also compare with four popular existing approaches– Barbosa et al. [33], Holmes and Murphy [55], Takuya and Masuhara [80] and Bajracharya et al. [32], 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 study.

- 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* [26], that captures the context code in the IDE, and recommends relevant exception handling code examples collecting from a remote web service [26], and the service can be leveraged by any IDE.

6.2 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 6.1). 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

Listing 6.1: Code under Development (i.e., Context Code)

```
//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
  }
```

triggered from within the *try* block and *suppresses* each of them, which clearly violates the second accepted principle[1] 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 designed handlers is highly error-prone and time-consuming. Second, exceptions

[1] https://www.ibm.com/developerworks/library/j-ejbexcept

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 [41], and thus leads to different hidden bugs and resource or security issues.

Listing 6.2: Recommended Code Example for Context Code in Listing 6.1

```
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();
   }
}
```

Now let us consider the code example in Listing 6.2 recommended by the proposed approach for the current programming context (i.e., code under development) in Listing 6.1. 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 not often aware of the exceptions which

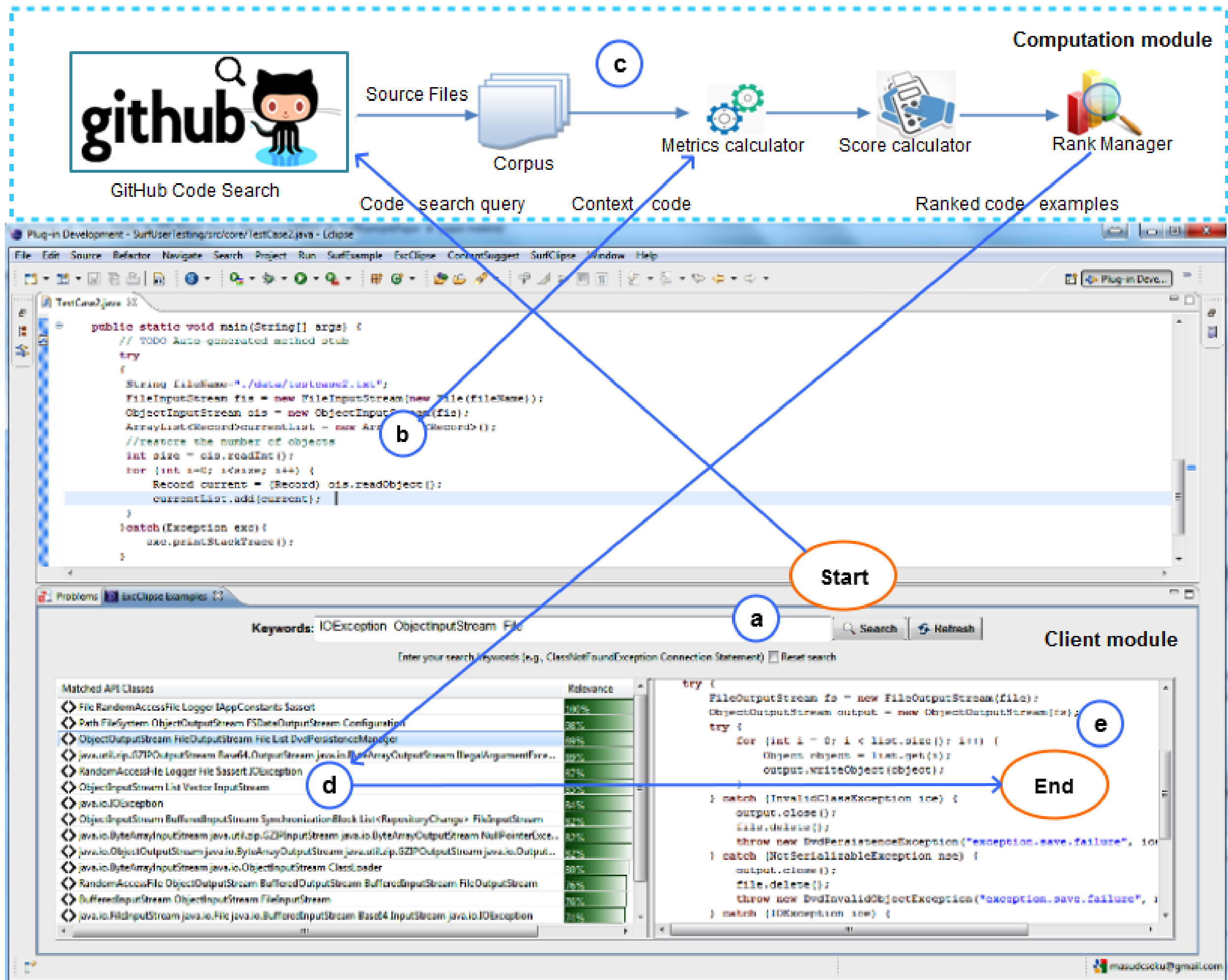

Figure 6.1: Schematic Diagram of Proposed Approach (SurfExample)

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 6.2). 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 [11, 12] 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 exception is a frequently misunderstood aspect of programming especially with applications of multilayer abstraction, and such exception handling code example can act as a helpful learning tool for the developer. The example is not necessarily meant for reuse; however, it can guide her towards effective handling in her application context through exemplary implementation.

6.3 Proposed Approach

Fig. 6.1 shows the schematic diagram of our proposed approach for exception handling code example recommendation. We package our solution into an easily accessible web service [26] and an Eclipse plugin [26]. In this section, we discuss different working modules of the solution, proposed metrics for relevance estimation between code segments as well as exception handler quality estimation, metric weight estimation techniques and ranking algorithms.

6.3.1 Working Modules

The proposed solution adopts a client-server architecture, and it has two working modules– *client module* (Fig. 6.1-(a, b, d, e)) and *computation module* (Fig. 6.1-(c)). The client module, an Eclipse plugin prototype [26], 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 6.4.2). It then sends the search query as well as the context code to the computation module. The computation module, hosted as a web service [26], collects code examples from *GitHub code repositories* using that search query and *GitHub code search API*, and develops a dynamic corpus (Fig. 6.1-(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 6.3.2) and ranking algorithms (Section 6.3.3). Once the ranked examples are returned from the computation module, the client module recommends the top 15 of them in the IDE (Fig. 6.1-(d, e)). The developer then can check the code examples and leverage for exception handling in her own programming context.

6.3.2 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.

Structural Relevance (R_{str})

Barbosa et al. [33] 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 [55] 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 [33, 55] 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. [67] 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* [14], 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 [33, 55] ignore the 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 [33, 55] 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 [67]. 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. 2.2, 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:

$$R_{str} = \alpha \times N + \beta \times \sum_{i=1}^{N} \frac{FAM_i}{FAQ_i} + \gamma \times \sum_{i=1}^{N} \frac{MIM_i}{MIQ_i} + \delta \times \sum_{i=1}^{M} DMT_i \quad (6.1)$$

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 6.3.4. 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.

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* [3] 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 [75], where we determine the *longest common subsequence* of tokens between the context code and the candidate code, and then normalize it as follows:

$$S_{ccm} = \frac{|S_{lcs}|}{|S_{total}|} \quad (6.2)$$

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:

$$R_{lex} = \lambda \times S_{cos} + \sigma \times S_{ccm} \quad (6.3)$$

Here λ and σ are the weights of the corresponding measures, and they are calculated using a machine learning technique involving logistic regression (Section 6.3.4).

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 [37]. 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 [37] 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 [28] 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:

$$Q_{ehc} = \mu \times R + \epsilon \times AHA + \kappa \times HCR \tag{6.4}$$

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 6.3.4). 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 [37] (i.e., code with too many handlers).

6.3.3 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 (6.5). The component scores belong to different ranges due to heterogeneous feature values, and each score is *normalized* between zero to one.

$$R_{total} = w_{str} \times R_{str} + w_{lex} \times R_{lex} + w_{ehc} \times Q_{ehc} \tag{6.5}$$

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 6.3.4. 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 6.2 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 6.1, the example in Listing 6.2 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. 6.2). 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 for the programming context in Listing 6.1.

6.3.4 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 [29] that returns a logistic regression based classifier model [17] which is validated with ten-fold cross validation. 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 [61]. 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 [61]. 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 [26].

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.

6.4 Experimental Design, Results and Validation

6.4.1 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 [20] 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 6.4.2), 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 code examples in total.

6.4.2 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 6.1), 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 6.1 is– *IOException URL.*

6.4.3 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 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 [11] 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 [26], and we use them as the benchmark examples to determine the performance of the proposed and existing approaches.

6.4.4 Performance Metrics

Our approach profoundly aligns with the research areas of information retrieval and recommendation systems. In order to evaluate our approach, we thus use a list of performance metrics from those areas [21] as follows:

Table 6.1: 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

Mean Average Precision at K (MAPK): *Precision at K* calculates *precision* at the occurrence of every relevant result in the ranked list. *Average Precision at K (APK)* averages the *precision at K* for all relevant results in the list for a query. *Mean Average Precision* is the mean of *average precision at K* for all queries.

$$APK = \frac{\sum_{k=1}^{D} P_k \times rel_k}{|RR|} \tag{6.6}$$

$$MAPK = \frac{\sum_{q \epsilon Q} APK(q)}{|Q|} \tag{6.7}$$

Here, rel_k denotes the relevance function of kth result in the ranked list, P_k denotes the precision at k^{th} result, and D refers to number of total results. RR is the set of relevant results for a query, and Q is the set of all queries.

Mean Precision (MP): *Precision* determines the percentage of the relevant results in the result list for a query. *Mean Precision* averages that percentage for all queries in the dataset.

Recall (R): *Recall* denotes the fraction of all the relevant results that are retrieved.

6.4.5 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 6.4.4). This section discusses the experimental results and the recommendation performance of our approach.

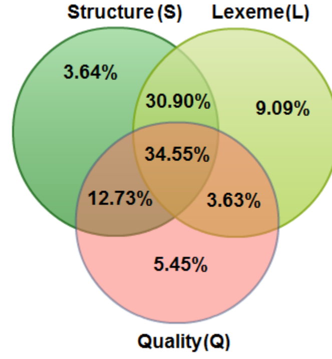

Figure 6.2: Result Distribution over Different Metrics

Table 6.2: Experimental Results on Different Aspects of Code

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

[1]Metrics for the top 5 results, [2]Metrics for the top 10 results, [3]Metrics for the top 15 results, [4]Percentage of correct examples recommended

Table 6.1 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 6.1, 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 [59, 72, 81] from the literature.

Fig. 6.2 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 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. 6.2, 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 6.2 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 6.2, 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 the exceptions in 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. 6.2.

Table 6.3: Comparison with Existing Approaches

Recommender	Metric	Top 5	Top 10	Top 15
Barbosa et al. [33]	MP	8.92%	6.92%	5.64%
	MAPK	**16.15%**	14.69%	13.72%
	TEH(65)	18(29)	25(45)	29(55)
	PEH	27.69%	38.46%	**44.62%**
	R	16.47%	25.57%	**31.25%**
Holmes and Murphy [55]	MP	6.15%	5.85%	5.03%
	MAPK	4.62%	2.31%	2.31%
	TEH(65)	16(20)	25(38)	31(49)
	PEH	24.62%	38.46%	**47.69%**
	R	11.36%	21.59%	**27.84%**
Takuya and Masuhara [80]	MP	8.31%	7.38%	5.54%
	MAPK	**21.54%**	20.51%	19.74%
	TEH(65)	22(27)	31(48)	31(54)
	PEH	33.85%	47.69%	**47.69%**
	R	15.34%	27.27%	**30.68%**
Bajracharya et al. [32]	MP	5.85%	4.31%	3.49%
	MAPK	8.46%	7.95%	6.41%
	TEH(65)	12(19)	18(28)	20(34)
	PEH	18.46%	27.69%	**30.77%**
	R	10.80%	15.91%	**19.32%**
Proposed Approach (local repo.) Structure(R_{str}) only	MP	**13.54%**	8.77%	7.18%
	MAPK	**21.80%**	19.87%	18.85%
	TEH(65)	**30**(44)	**33**(57)	**37**(70)
	PEH	46.15%	50.77%	56.92%
	R	**25.00%**	**32.38%**	**39.77%**
Proposed Approach (local repo.) Structure(R_{str}), Content(R_{lex}), and Quality(Q_{ehc})	MP	**13.85%**	9.23%	7.90%
	MAPK	**30.64%**	27.44%	25.90%
	TEH(65)	**31**(45)	**34**(60)	**40**(77)
	PEH	**47.69%**	52.31%	**61.54%**
	R	**25.56%**	**34.09%**	**43.75%**
Proposed Approach (GitHub search) Structure(R_{str}) and Content(R_{lex})	MP	**27.99%**	17.99%	13.44%
	MAPK	**43.07%**	38.69%	37.33%
	TEH(65)	45(91)	49(117)	**53**(131)
	PEH	69.23%	75.38%	**81.54%**
	R	**51.70%**	**66.48%**	**74.43%**
Proposed Approach (GitHub search) Structure(R_{str}), Content(R_{lex}), and Quality(Q_{ehc})	MP	**31.07%**	18.62%	13.85%
	MAPK	**41.92%**	39.92%	38.64%
	TEH(65)	**48**(101)	**53**(121)	**56**(135)
	PEH	73.85%	81.54%	**86.15%**
	R	**57.39%**	**68.75%**	**76.70%**

6.4.6 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. [33], Holmes and Murphy [55], Takuya and Masuhara [80] and Bajracharya et al. [32]. 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. [33] 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 6.3 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 rest three 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 [55] 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 [80] use *cosine similarity* in order to determine relevance between two code examples. Bajracharya et al. [32] 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 [32] are not sufficient enough to request for exception handling code examples, and we use the queries (Section 6.4.2) 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 6.3, 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 6.2, our approach also performs significantly better than those approaches without using the handler quality 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.

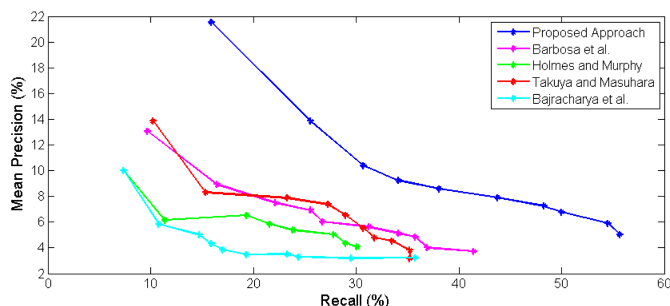

Figure 6.3: Mean Precision vs. Recall Curves

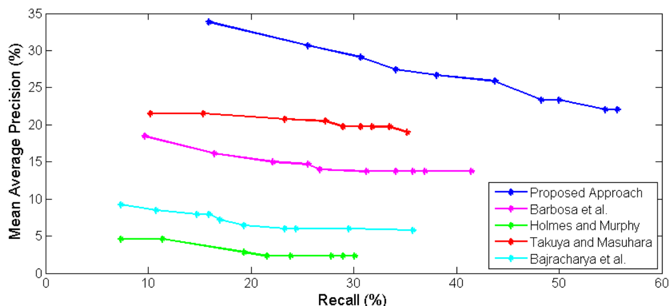

Figure 6.4: Mean Average Precision vs. Recall Curves

As shown in the schematic diagram in Fig. 6.1-(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 6.3, 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. 6.3 and *mean average precision-recall* curve in Fig. 6.4 also show 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, the proposed approach outperforms all other approaches in the experiments.

6.5 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 the first author has professional development experience (details in Section 6.4.3).

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 lots of time and efforts, and we covered most of the well known standard Java exceptions [26] 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.

6.6 Related Work

Exception handling is not a new topic, and there exists a good number of studies [33, 38, 41, 47, 49, 74]. Barbosa et al. [33] propose an approach to recommend exception handling code by exploiting three heuristics about structural facts in the code. The approaches by Holmes and Murphy [55], Takuya and Masuhara [80] and Bajracharya et al. [32] 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 6.4.6.

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. [41] 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 [74] propose another static analysis approach that identifies different possible exceptional flows in the application program and helps the developer to understand 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 [33]. 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 [33]. Garcia et al. [47] 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. [77] 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.

6.7 Summary

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, and conduct experiments with 65 exceptions (and their context code) and 4,400 code examples. Our approach can recommend relevant examples for 86.15% of the exceptions with a maximum of 41.92% *mean average precision* and 76.70% *recall.* We also compare with four existing approaches, where our approach outperforms them in all performance metrics. While our experiments show that the general-purpose code recommendation approaches are not satisfactorily applicable for the recommendation of exception handling code, in this study, 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. While each of the proposed approaches in the four studies (Chapter 3, Chapter 4, Chapter 5 and Chapter 6) is extensively evaluated and validated in isolation, we were interested to study whether the developers would find them useful for problem solving when they are integrated. We thus integrate all four approaches into an Eclipse plugin, and conduct a user study on the integrated approach in Chapter 7. Furthermore, the primary purpose of this user study was to explore the usability of the integrated approach and not an absolute evaluation. We extensively evaluated each of the individual contributions in the corresponding chapters of the thesis.

Chapter 7

A Comparison between Proposed Methods and Traditional Ones: A Task-Based User Study

Each of the proposed approaches in the four studies–(a) context-aware meta search engine (Chapter 3), (b) context-aware search query recommendation (Chapter 4), (c) context-aware page content suggestion (Chapter 5), and (d) context-aware code example recommender for exception handling (Chapter 6), is extensively evaluated using experimental data. Their performance is also extensively validated against relevant approaches from the literature, and using several mini user studies, and they demonstrate promising results. However, in addition, we are interested to investigate the usability and the effectiveness of an approach that integrates all four of them using a task-oriented user study. This chapter discusses detailed design and findings of our conducted user study on the integrated approach that supports different search-related activities in dealing with programming errors and exceptions.

The rest of the chapter is structured as follows– Section 7.1 focuses on our developed prototype for the integrated approach–*ExcClipse*, and Section 7.2 discusses tasks, participants, data collection techniques and other details of the study design. Section 7.3 discusses the study execution, Section 7.4 analyzes the results and reports the findings from the study, Section 7.5 identifies the potential threats to validity of the study, and finally Section 7.6 summarizes the chapter.

7.1 ExcClipse

Each of our proposed approaches is implemented as an Eclipse IDE plugin prototype, and we develop four such prototypes–*SurfClipse, QueryClipse, ContentSuggest* and *SurfExample*. However, each prototype focuses on a particular recommendation service (e.g., search query recommendation), and we are interested to investigate the potential of a plugin prototype that combines all four recommendation services using a user study. We integrate all four prototypes into one, called *ExcClipse*, and this section provides an overview of the integrated prototype. *ExcClipse* provides two types of service–*web page search* and *code example search* as follows:

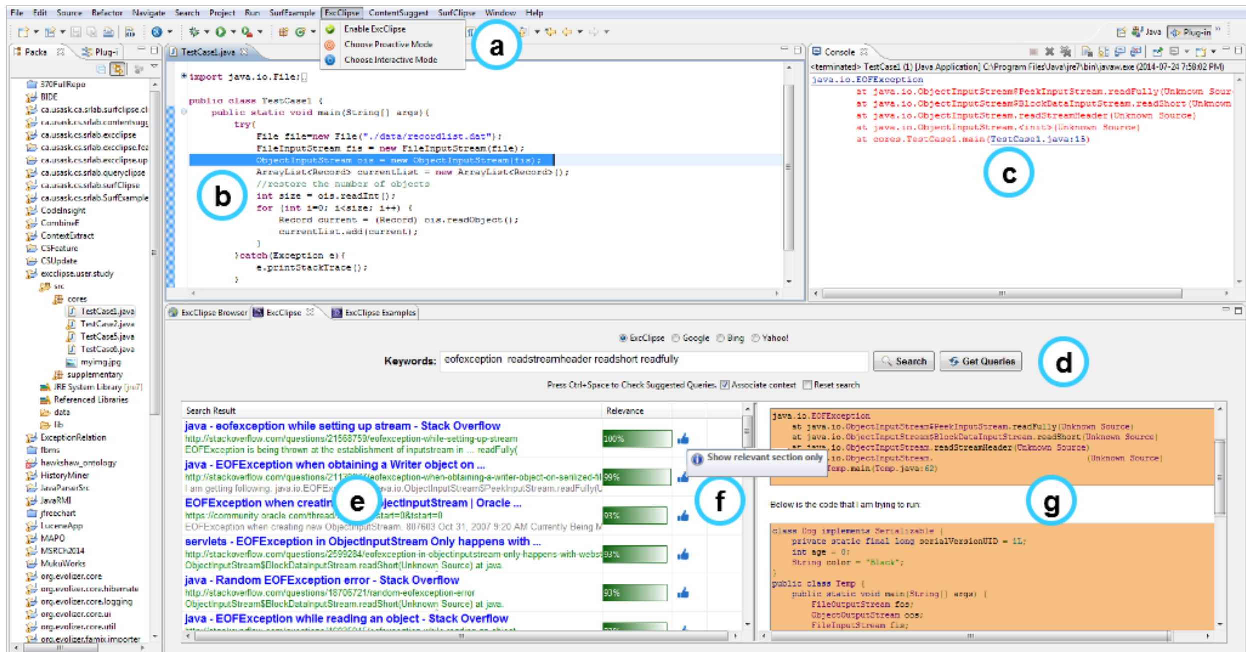

Figure 7.1: ExcClipse Web Search

7.1.1 ExcClipse Web Search

Fig. 7.1 shows the user interface of *ExcClipse*, where we contribute in (a) setup and configuration panel, (d) web search panel, (e) search result panel, and (g) relevant content panel. This subsection discusses different technical features and supports provided by the plugin prototype.

(1) Working Modes: *ExcClipse web search* works in two modes–*interactive* and *proactive*. In case of *interactive* mode, a user (e.g., a developer) generally initiates the web search by selecting an exception from *Console View* in the IDE or using a search query (representing the exception and its context) from the recommendation list, whereas the plugin prototype itself initiates the search process in case of *proactive* mode. Once the prototype is properly installed, it provides several main menu (e.g., Fig. 7.1-(a)) and context-menu (e.g., Fig. 7.3) based command options, which can be used to initiate the plugin environment and to change the working modes.

(2) Automated Supports with Search Queries: Both the context code (e.g., Fig. 7.1-(b)) that triggers an exception, and the stack trace (e.g., Fig. 7.1-(c)) reported by the IDE contain overwhelming information, and developers often face difficulties in choosing a suitable search query from such information. *ExcClipse* provides an automated support in this regard, and helps them choose queries from a list of recommended queries. It analyzes both stack trace and context code of the exception, and recommends a ranked list of five suitable search queries for the exception (Fig. 7.2). One then can either select a query or develop a customized query by leveraging the tokens in the selected query for web search.

(3) Context-Aware Web Search: *ExcClipse* provides three options to conduct web search within the IDE–*proactive search*, *context-menu based search (interactive mode)* and *keyword search (interactive mode)*.

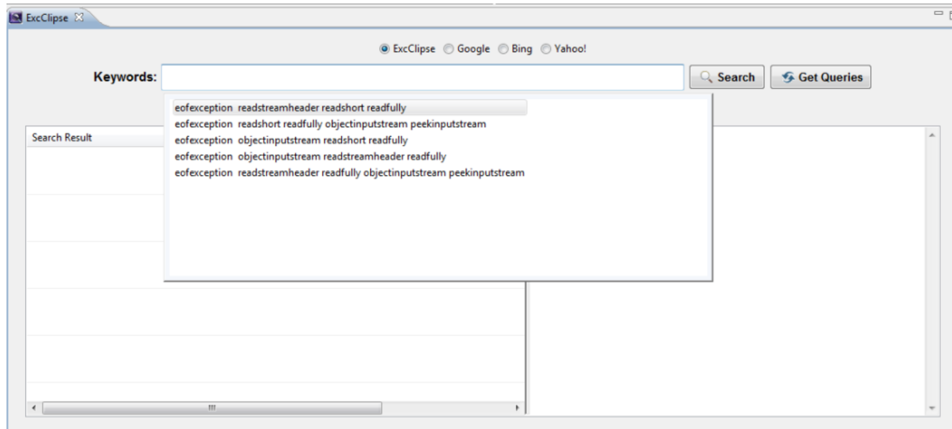

Figure 7.2: Web Search Query Recommendation (Interactive Mode)

Each of these options requires the developer to interact differently with the IDE although they follow the same working principles in the background for collecting search results.

Proactive Search: When *ExcClipse* is set to *proactive mode*, it automatically detects an encountered exception in the IDE. In this mode, the prototype constantly monitors the *Console View* (e.g., Fig. 7.1-(c)) for a stack trace using regular expressions. Upon detection, it collects other details– an auto-generated search query and the context code (e.g., Fig. 7.1-(b)) of the exception, and initiates the search.

Context-Menu Based Search: The prototype provides a context-menu based web search option, and a developer can literally select any phrase in the IDE (from *Editor View* and *Console View*), and perform web search. More importantly, she can select the exception in the *Console View*, and conduct the search (e.g., Fig. 7.3). Once initiated, the plugin captures necessary details from the IDE, performs the search, and collects the results.

Keyword Search: Given that a user might be interested in refining the auto-generated search query or in a more traditional way of search, the plugin provides a keyword-based search feature (e.g., Fig. 7.1-(d)). The search is complemented with search query suggestion through auto-completion. The user can also configure whether the search should be a *keyword matching only* (i.e., does not refine the results against the *exception context*) or a *context-aware* one through *Associate context* option (e.g., Fig. 7.1-(d)).

(4) Search Results & Browsing: Once a search request is made for an exception, the plugin collects results in a non-intrusive way (i.e., without freezing the IDE), and displays them within the IDE (e.g., Fig. 7.1-(e)). It visualizes the relative relevance of each result page through visualization (e.g., Fig. 7.1-(f)), which helps one to choose the right (most relevant) page for browsing. It also facilitates browsing of the result page in the following two ways:

Relevant Section(s) Browsing: Once a developer chooses to check only the relevant content of a result page (e.g., Fig. 7.1-(f)), the plugin analyzes all the sections in the page, discards noisy sections and also some noisy elements, determines relevance of each section against the exception and its context in the IDE, and then returns the most relevant section(s) from the page ((e.g., Fig. 7.1-(g))). The idea is to help the developer not only in determining the relevance of a result page (before actually browsing the page) but also

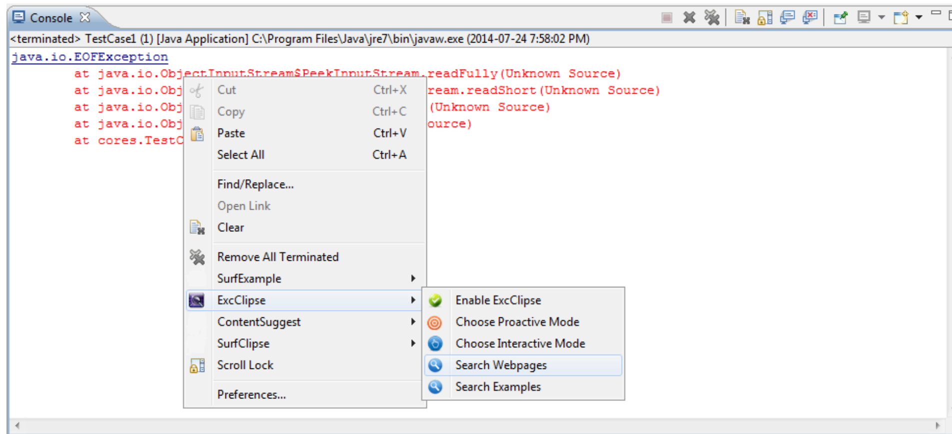

Figure 7.3: Context-Menu Based Search (Interactive Mode)

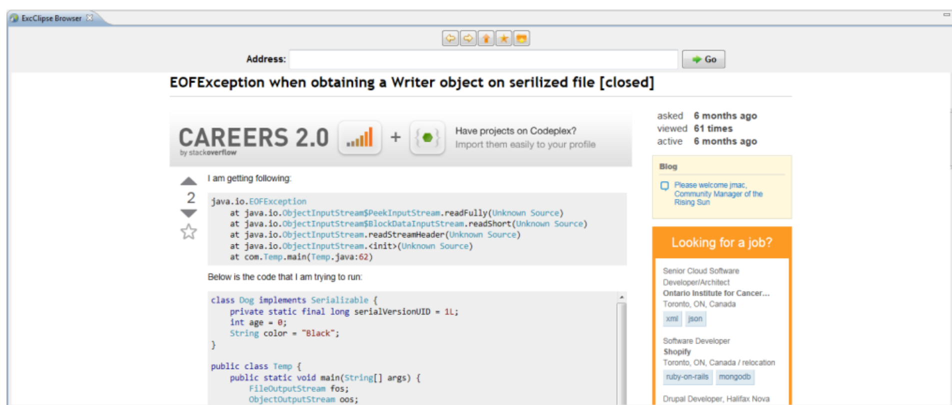

Figure 7.4: Web Page Browser

in finding solutions for the exception through consulting less information and putting less cognitive effort.

Actual Page Browsing: Once a developer is convinced of the relevance of a result page through the checking of relevant section(s), she can consult the actual page for further analysis. The plugin prototype facilitates the browsing using a customized browser widget (e.g., Fig. 7.4).

7.1.2 ExcClipse Code Example Search

Fig. 7.5 shows the user interface of *ExcClipse* for code example search, where we contribute in (b) search panel, (c) search result panel and (d) code preview panel. This subsection discusses different technical features of the plugin associated with code example search.

(1) Context-Aware Code Search: *ExcClipse* provides two options to conduct code example search within the IDE–*context-menu based search* and *keyword search*. Both options require a developer to perform such interactions with the IDE which are similar to that of *ExcClipse web search*.

Context-Menu Based Code Search: The prototype provides a context-menu based code search option, and a developer can select any *exception class* in the IDE (from *Editor View*) and perform code search. More

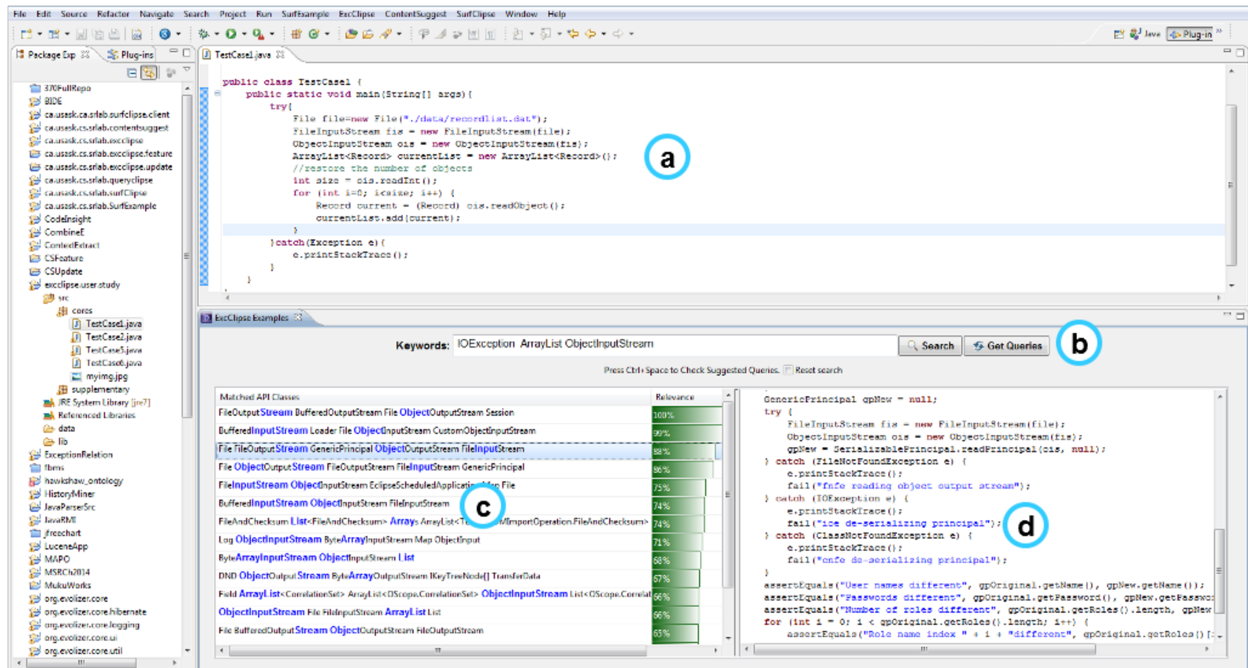

Figure 7.5: ExcClipse Code Example Search

importantly, she can select a *class method* under development in the IDE (i.e., context code) and conduct the search (e.g., Fig. 7.6). Once initiated, the plugin captures necessary details– an auto-generated search query and the context code (e.g., Fig. 7.5-(a)), and returns a list of code examples that are relevant to the context code and handle an exception of interest.

Keyword Based Code Search: The plugin prototype provides a keyword-based code search feature (e.g., Fig. 7.5-(b)), and a developer can search with a customized query by choosing suitable keywords from the context code. The search is also complemented with search query recommendation (e.g., Fig. 7.7) through auto-completion, and the queries are recommended by analyzing both the context code (e.g., Fig. 7.5-(a)) and the exceptions that might trigger from such code. Thus one can just select a query from the recommendation list and perform code example search for exception handling.

(2) Code Search Results & Browsing: Once a code search request is made for the context code in the IDE, the plugin collects relevant code examples and displays them within the IDE (e.g., Fig. 7.5-(c)). It highlights the matched query keywords and visualizes the relative relevance of each resultant example, which help one to choose the right (most relevant) example for browsing. Once an example is selected, the code preview panel (e.g., Fig. 7.5-(d)) helps one not only to analyze the relevance of the example against the context code (e.g., 7.5-(a)) in close proximity but also to check the quality of the handlers for the exception of interest (e.g., *IOException* in case of the context code in Fig. 7.5-(a)).

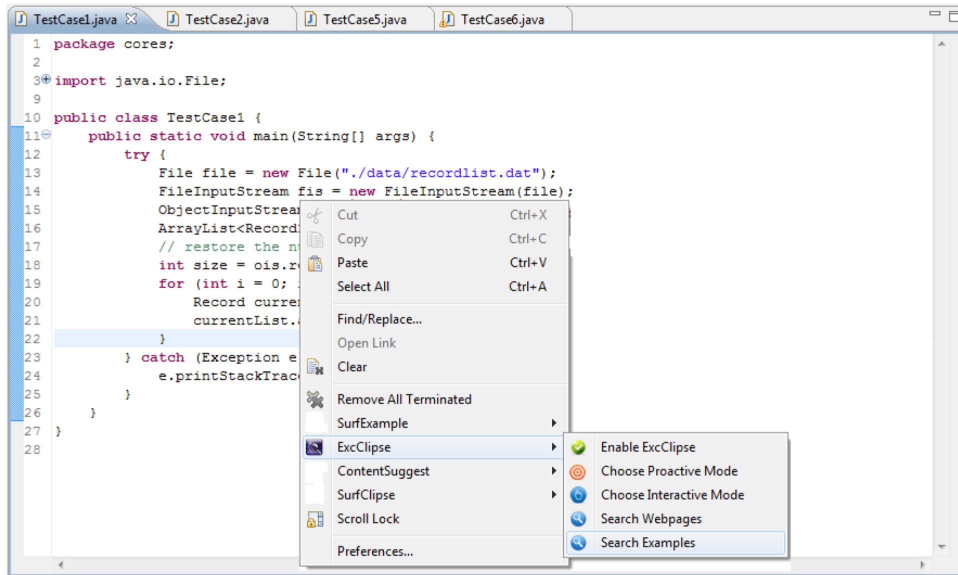

Figure 7.6: Context-Menu Based Code Search

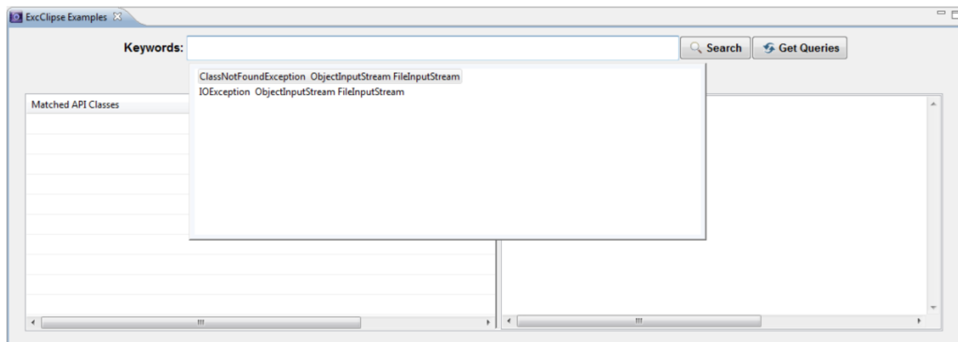

Figure 7.7: Code Search Query Recommendation

7.2 Design of User Study

In order to capture user feedback on *ExcClipse* and to contrast with traditional alternatives, we conduct a task-oriented user study. The study requires a list of tasks, a group of participants and a list of data collection as well as evaluation tools and techniques. This section discusses the detailed design of our conducted study.

7.2.1 Tasks Design

In the user study, we contrast between the search related supports of *ExcClipse* and those of traditional approaches (e.g., keyword-based web search engines, code search engines), where we choose a list of four tasks associated with web search and code example search. The tasks are not necessarily of equal granularity and difficulty; however, we ensure that they are specific enough for easier evaluation or comparative analysis and important enough to reflect the potential of an approach.

T1: Prepare a search query for an encountered exception by analyzing its stack trace and the code that triggers the exception.

Target Usage: search query recommendation

T2: Perform web search using the query and collect one or more relevant web pages that most probably contain solutions for a given exception.

Target Usage: context-aware meta search

T3: Analyze if the recommended section (by *ExcClipse*) of a web page is actually the most relevant part of the page for the exception at hand.

Target Usage: relevant page section recommendation

T4: Prepare a search query for the code under development in the IDE, and collect a list of relevant code examples that can help in improving the exception handlers in the target code.

Target Usage: relevant code example search for exception handling

7.2.2 Exception Test Cases

We consider four exception test cases–EC_1, EC_2, EC_3 and EC_4 for the study, where each case contains an exception, a stack trace and a context code segment. The exceptions are generally associated with file (e.g., *EOFException*) or image (e.g., *IIOException*) manipulation, Java reflection (e.g., *IllegalAccessException*) and network connections (e.g., *UnknownServiceException*), and all the cases can be found in Appendix A. In order to solve and handle each case, a participant performs the above four tasks (Section 7.2.1) using two different environments–*traditional* (i.e., web search engines, code search engines) and *ExcClipse*, and we capture the participant's experiences and feedback details for comparative analysis.

7.2.3 Study Participants

We choose six graduate research students from Software Research Lab, University of Saskatchewan, as the participants for the study. Each of the participants has at least three years of Java programming and a substantial amount of problem solving experiences that involve both web search and code search. Three of the participants also have prior professional software development experience.

7.2.4 Study Data Collection

We apply two techniques–*simple observation* and *questionnaire* in order to capture the problem solving experience as well as the feedback from the participants respectively. While in the first case, we observe how a participant interacts with a search provider (i.e., both traditional search engines and *ExcClipse*) for problem solving, in the second case, we collect direct feedback on the system from the participant. The observation checklist and the questionnaire for the study can be found in Appendix B.

7.3 Running the Study Session

Each study session generally takes about 1–1.5 hour, and it consists of three phases–*training phase, execution phase* and *evaluation phase*. This section discusses briefly about each of those phases as follows:

7.3.1 Training Phase

We develop a video tutorial[1] of ExcClipse that introduces the prototype to the participants. It also shows how each of the study tasks (Section 7.2.1) can be accomplished using ExcClipse. Each study session involves one participant, and we use the video to train the participant during training session. We also occasionally let the participants to play around for a few minutes to get themselves introduced with the prototype. The phase lasts for about 15-20 minutes.

7.3.2 Execution Phase

In this phase, a participant performs the four tasks for each of the exception test cases using ExcClipse and traditional search providers (e.g., Google, Bing and Yahoo) in two limited sessions– Session I and Session II. In order to avoid bias in findings from the study, we categorize the test cases, the participants and even the execution sessions into multiple groups. Table 7.1 shows the detailed plan of the conducted study. For example, the participants are divided into two groups– *Group A* (P_1, P_4, P_5) (i.e., blue coloured) and *Group B* (P_2, P_3, P_6). Participants from Group A solve and handle two exception cases–EC_1 and EC_2 using traditional means and the other two cases–EC_3 and EC_4 using ExcClipse, which is vice versa for the participants from Group B. We also organize the execution sessions such a way so that three participants (i.e., Group I) use the traditional means for the study in session I, whereas the rest three (i.e., Group II) use ExcClipse for the same purpose in session I. Table 7.2 shows the organization of sessions and corresponding participants. During execution phase, we observe the problem solving practices of the participants and occasionally ask different clarification questions in order to capture their experience in the form of observation checklists (Section B.1.1 and Section B.1.2). This phase lasts about 40-50 minutes.

7.3.3 Evaluation Phase

In this phase, each participant fills in the questionnaire (Section B.2) that contains 15 questions related to usability, effectiveness, efficiency, look and feel and other details of the two search providers – ExcClipse and traditional search engines. While recording responses, the participants contrast between the two alternatives based on their experiences from the execution phase, and we also collect their qualitative comments and suggestions. This phase lasts for about 10-15 minutes.

[1] https://www.youtube.com/watch?v=-FBON-2qhfA

Table 7.1: Study Session Plan

Exception Test Cases	Participants					
	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	$\mathbf{P_5}$	$\mathbf{P_6}$
$\mathbf{EC_1/T}$	$\odot$			$\oplus$	$\odot$	
$\mathbf{EC_1/E}$		$\oplus$	$\odot$			$\odot$
$\mathbf{EC_2/T}$	$\odot$			$\oplus$	$\odot$	
$\mathbf{EC_2/E}$		$\oplus$	$\odot$			$\odot$
$\mathbf{EC_3/T}$		$\odot$	$\oplus$			$\oplus$
$\mathbf{EC_3/E}$	$\oplus$			$\odot$	$\oplus$	
$\mathbf{EC_4/T}$		$\odot$	$\oplus$			$\oplus$
$\mathbf{EC_4/E}$	$\oplus$			$\odot$	$\oplus$	

$\mathbf{EC_i/T}$=Solved using traditional approach

$\mathbf{EC_i/E}$=Solved using ExcClipse

$\odot$=Done in session I, $\oplus$=Done in session II

Table 7.2: Study Execution Sessions

Session I	**Session II**	**Participants**
Traditional approach	ExcClipse	Group I (P_1, P_2, P_5)
ExcClipse	Traditional approach	Group II (P_3, P_4, P_6)

7.4 Result Analysis and Discussions

We analyze both the observation data collected during execution phase and the feedback from the participants collected during evaluation phase, and contrast ExcClipse with traditional means for different search based activities associated with programming errors and exceptions. This section reports the findings from our comparative analysis.

7.4.1 Evaluation Metrics

We contrast ExcClipse with traditional search engines for different features associated with search-related activities, and we apply two widely used metrics for evaluation– *Average Rating* and *Mann-Whitney U-Test* [63, 64, 83]. While using the first metric, it can be shown whether two lists of ratings are different in terms of their central values which might not be enough for effective comparison all the time, the second metric determines whether the lists are significantly different from each other. We define the both metrics as follows:

Average Rating: It averages a list of ratings, where each rating is restricted by a certain interval. For example, in our study, while a rating "1" refers to the lowest rating, "10" refers to the highest rating. We calculate average rating for every single feature separately as well as the entire system for comparative analysis.

Table 7.3: Severity, Frequency and Relevance Scales

Difficulty Level	Scale	Inconvenience Level	Scale	Quality Level	Scale	Relevance Level	Scale
Very hard	5	Very problematic	5	Excellent	5	Highly relevant	5
Hard	4	Problematic	4	Good	4	Relevant	4
Neither hard nor easy	3	Does not matter	3	Somewhat good	3	Somewhat relevant	3
Easy	2	Helpful	2	Neither good nor bad	2	Hardly relevant	2
Very easy	1	Very helpful	1	Not good	1	Not relevant at all	1

Table 7.4: Participants' Ratings for Motivating Factors behind ExcClipse

Motivating Factor	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	$\mathbf{P_5}$	$\mathbf{P_6}$	Avg.	Comment
Context-switching between IDE and browser (Inconvenience)	4	3	5	5	5	3	**4.17**	Problematic
Manual search query formulation (Difficulty)	3	4	3	4	3	5	**3.67**	Hard
Merit or prospect of a meta search based approach (Quality)	5	4	5	4	4	5	**4.50**	Excellent
Traditional search query (Relevance)	3	3	4	3	3	3	3.17	Moderate
Checking web page relevance (Difficulty)	4	3	4	4	5	3	**3.83**	Hard

$\mathbf{P_i}$=Ratings by i^{th} participant

Mann-Whitney U-Test[2]: It is a non-parametric statistical test that compares between two independent sets of ordinal measures. We use the test to understand whether the ratings of the participants for ExcClipse significantly differ from that of traditional alternative as a whole or for different search related individual features. The test outputs two measures– U and p-values, and we consider a widely used significance level of 0.0500 as suggested by relevant existing studies [63, 64, 83]. Thus if p-value $<$0.0500, we consider the rating difference is significant and vice versa.

7.4.2 Motivating Factors for ExcClipse

We identify five important motivating factors behind different approaches proposed in our thesis, and we were interested to check the perceptions of the participants about them. We set up appropriate scales for *difficulty* (i.e., how difficult a task is?), *inconvenience* (i.e., how inconvenient a situation or scenario is?), *quality* (i.e., how good an approach or an idea is?) and *relevance* (i.e., how much relevant a page is?) in Table 7.3, and collect responses from the participants during evaluation phase by applying those scales. We basically use the first five questions in the questionnaire (Section B.2) to capture such responses from the participants. Table 7.4 shows the responses from the six participants, and we average the numerically transformed responses for analysis. From Table 7.4, we note that context-switching between IDE and browser during web search is found distracting and inconvenient for most of the participants, and formulation of an effective search query about a programming error or an exception is also a non-trivial task for them. The participants also find checking relevance of a web page against an exception and its context generally hard. According to the

[2]http://www.socscistatistics.com/tests/mannwhitney

Table 7.5: Tool Features for Evaluation

Feature	Functionality	Notation
Support for query formulation	Web search	F_1
Accuracy & effectiveness of results	Web search	F_2
Post-search content analysis	Web search	F_3
Support for query formulation	Code search	F_4
Relevance and accuracy of results	Code search	F_5
Usability	Overall	F_6
Efficiency	Overall	F_7
Visualization support	Web search	F_8
Visualization Support	Code search	F_9

recorded responses, they find the recommended queries by the search engines relevant sometimes, and they highly appreciate the idea of a meta search based approach that collects top-ranked results from multiple popular search engines (e.g., Google, Bing and Yahoo), and then ranks them with a reliable (e.g., context-aware) ranking. Our previous experiments (Section 3.3.7, Table 3.5) in the first study also show that the approach has the potential indeed.

7.4.3 Comparison of ExcClipse with Traditional Search Providers

Tool Feature Specific Comparison

We choose nine individual features which are distinct for each of the search providers– *ExcClipse* and traditional search engines. During evaluation phase, these features are evaluated by the participants for both search providers, and they are also used to contrast *ExcClipse* with traditional search engines for different search-related activities. Table 7.5 shows those features as well as their shorthand notations that are used to refer to the features in the rest of the chapter. We also categorize the features into three– *web search features, code search features* and *non-functional features* as follows:

Web Search Features: In web search, a user generally chooses a few keywords as a search query, and collects a list of relevant web pages. We compare the support by ExcClipse for web search with that of traditional search engines (e.g., Google) using three features– *query formulation (F_1), effectiveness of search (F_2)* and *post-search content analysis (F_3)*. Fig. 7.8 shows average ratings from the participants for both counterparts, and we note that ExcClipse is relatively rated higher for each of the features. We also perform Mann-Whitney U- test (Table 7.6) and experience that the ratings for ExcClipse are significantly higher than that of the traditional search engines.

Code Search Features: In code search, a user chooses a few program tokens (e.g., class name, method name) as a search query, and collects a list of relevant code examples. We consider the next two features in Table 7.5– *code search query formulation (F_4)* and *relevance and accuracy of code examples (F_5)* for

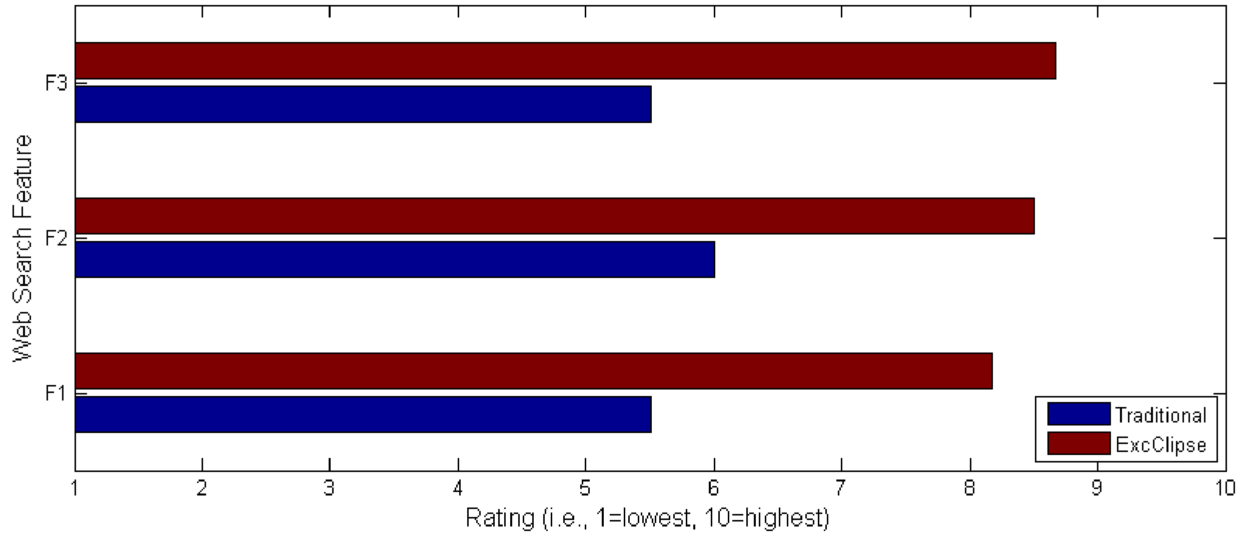

Figure 7.8: Ratings of Web Search Features

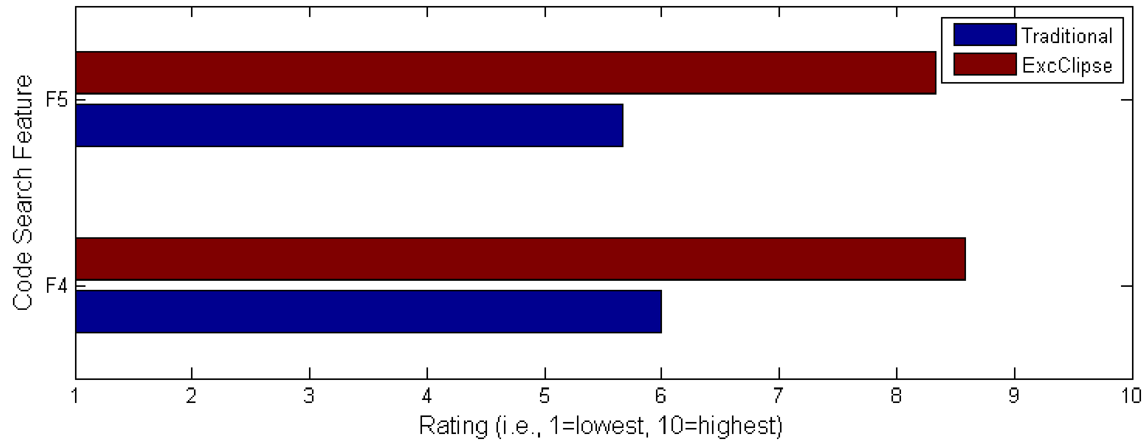

Figure 7.9: Ratings of Code Search Features

contrasting ExcClipse with traditional means for the support with code example search. Fig. 7.9 shows the average ratings for both approaches, and we get findings similar to that with web search features.

Non-functional Features: While the above features mostly focus on functional aspects of the search providers, the features from the third category are related with non-functional aspects such as usability, look and feel and so on. We also contrast ExcClipse with traditional search engines on those non-functional features (F_6, F_7, F_8 and F_9), and experience that our prototype is preferred over traditional means by the participants. From Table 7.6, we also note that the ratings for ExcClipse are significantly higher than that of its counterpart.

Tool level Comparison

We also consider the ratings from participants for nine individual features of each search provider all at once, and compare ExcClipse with traditional search engine. We average all the ratings of each search provider

Table 7.6: Rating of Tool Features by Participants

Feature	Approach	P_1	P_2	P_3	P_4	P_5	P_6	U-value	p-value	RD
F_1	Traditional	7	5	5	5	7	4	1	0.0083	S
	ExcClipse	8	9	9	8	8	7			
F_2	Traditional	7	6	8	5	6	4	1.5	0.0105	S
	ExcClipse	8	9	9	9	8	8			
F_3	Traditional	5	5	8	4	5	6	1.5	0.0105	S
	ExcClipse	9	10	9	9	7	8			
F_4	Traditional	7	4	6	6	6	7	0	0.0051	S
	ExcClipse	9	10	9	8	7.5	8			
F_5	Traditional	7	6	5	5	5	6	0.5	0.0065	S
	ExcClipse	9	9	9	8	8	7			
F_6	Traditional	6	6	5	6	6	6	0	0.0051	S
	ExcClipse	9	10	10	8	7	7			
F_7	Traditional	5	6	6	7	5	6	0.5	0.0065	S
	ExcClipse	9	9	8	9	8	7			
F_8	Traditional	6	5	8	6	6	7	3.5	0.0251	S
	ExcClipse	7	10	9	10	7	8			
F_9	Traditional	5	5	7	6	7	7	0	0.0051	S
	ExcClipse	8	10	9	8	8	9			
All Features	**U-value**	2	0	1.5	0	3	6	-	-	-
	p-value	0.0007	0.0004	0.0006	0.0004	0.0010	0.0027			
	RD[1]	S	S	S	S	S	S			

Traditional=Google (i.e., web search engine) and GitHub (i.e., code search engine)

Significance level=0.0500, **S**=Significant, **I**=Insignificant, **RD**=Rating Difference, [1]Difference between all ratings for traditional approach and ExcClipse respectively by a participant

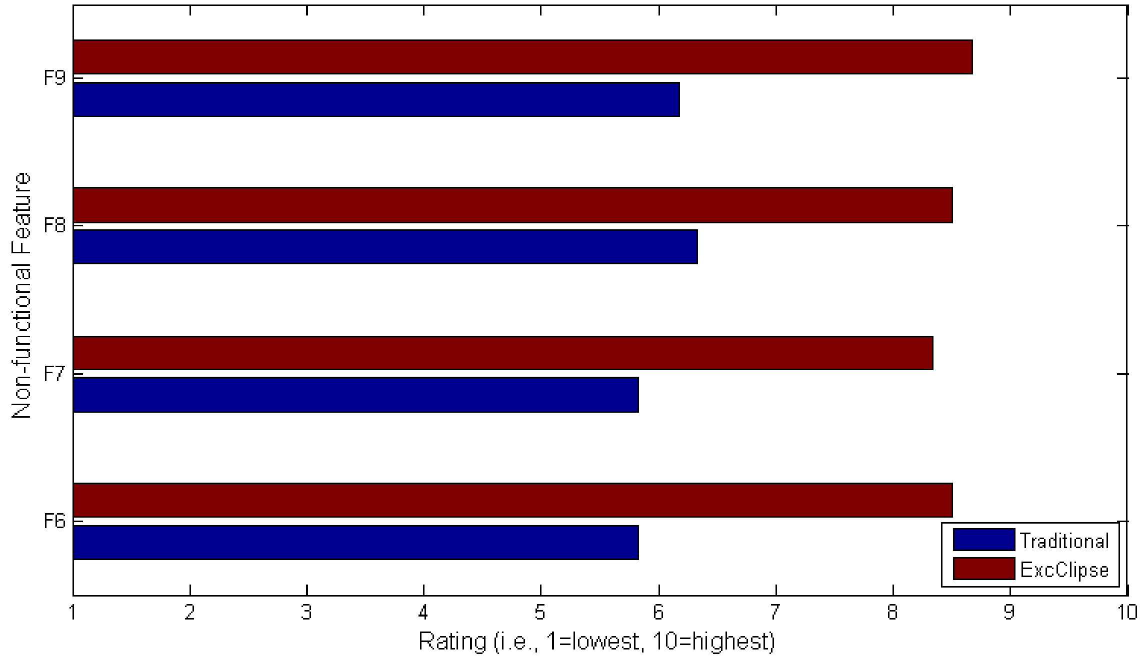

Figure 7.10: Ratings of Non-functional Features

from each participant and contrast the average ratings. The bar chart in Fig. 7.11 clearly shows that ExcClipse is highly rated compared to the traditional search engine by each of the participants. While the search engine receives a maximum average rating of 6.44 on the scale from one (i.e., lowest rating) to ten where ten being the highest, ExcClipse enjoys a maximum rating of 9.56 on the same scale. In order to check whether the ratings for ExcClipse are significantly higher than that of traditional search engine, we conduct Mann-Whitney U-test on both sets of ratings from the same participant. According to the last row of Table 7.6, the ratings for our prototype are significantly higher for each of the participants.

In order to check whether the participants do indeed agree on using ExcClipse, we compare their ratings using Mann-Whitney U-test as well. Table 7.7 reports the findings of that comparative analysis. Among $^6C_2$=15 participant pairs (i.e., total pairs generated from six participants), we find six pairs whose ratings are not significantly different. Thus the participants in those pairs provide almost similar ratings. We investigate the possible correlation between these matched ratings and the participant grouping in Table 7.1 and Table 7.2. For example, Fig. 7.12 visualizes the rating agreements among different participants (using connecting edges) of different groups– Group A and Group B, Group I and Group II. From Fig. 7.12, we do not notice much regular patterns among the agreements such as intra-group agreement or inter-group agreement except one instance. We experience a strong rating agreement between participant one (P_1) and two other participants (P_3 and P_6) in both groupings. Each of the participants took part in the study alone and rated independently, and the inter-group matchings (e.g., $P_1 - P_3$, $P_3 - P_4$, $P_5 - P_6$) reflect the confidence on the rating scale as well as our prototype. On the other hand, the lack of intra-group agreement indicates that the rating agreements are not biased by grouping patterns or any other relevant factors. It also should be

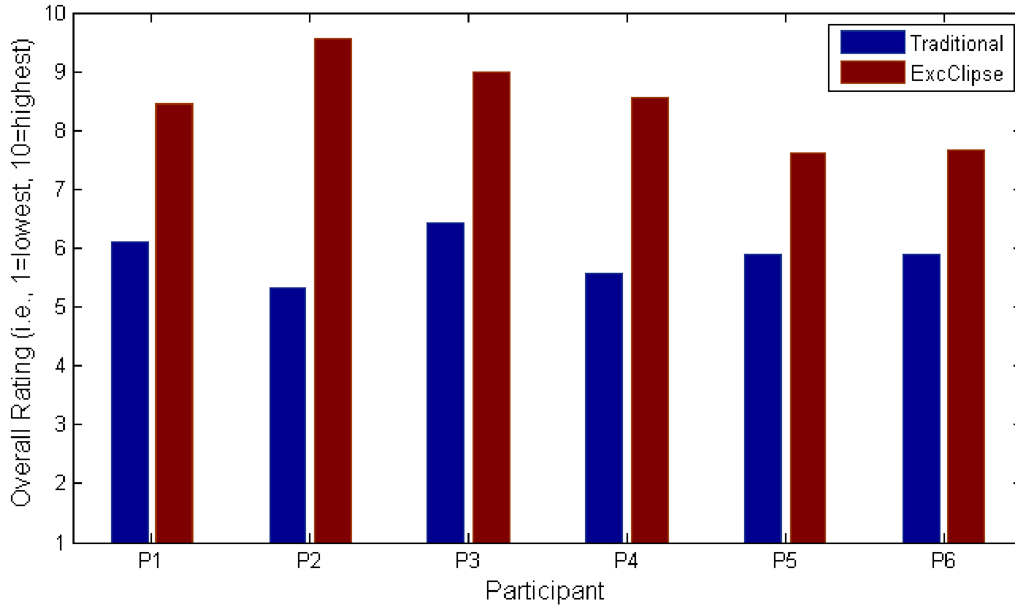

Figure 7.11: Overall Ratings from Participants (Difference is Significant, U=0, p=0.0051)

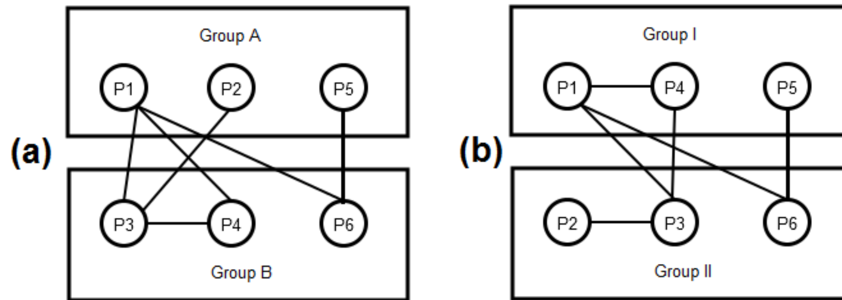

Figure 7.12: Agreement among Ratings from Participants (i.e., Edges refer to Agreements)

noted that the ratings from each of the remaining nine participant-pairs differ significantly. While we can speculate that the different factors such as programming experience of the participants, display settings (e.g., monitor), study environment (e.g., lab, resident) are contributing factors to such finding, the study should be conducted in a more controlled environment to explain the finding properly.

Comparison using Observed Responses

During execution phase, we observe how each participant searches for relevant web pages or code examples using both traditional search engines and ExcClipse, and record certain observations. We also occasionally ask the participants certain clarification questions at the end of the phase in order to confirm our observations. Table 7.8 shows 15 such of our recorded observations (i.e., questions), where we use numbers for numerical observations and scales for dichotomous observations. In case of numerical observations, we note that ExcClipse requires a participant to put relatively less efforts for *search query formulation* and *retrieval*

Table 7.7: Overall Rating Difference among Participants

$\mathbf{P_A}$	$\mathbf{P_B}$	**U-value**	**p-value**	**RD**	$\mathbf{P_A}$	$\mathbf{P_B}$	**U-value**	**p-value**	**RD**
P_1	P_2	10	0.0080	S	P_1	P_3	24	0.1584	I
P_1	P_4	40	1.0000	I	P_1	P_5	15	0.0271	S
P_1	P_6	18.5	0.0574	I	P_2	P_3	20.5	0.0854	I
P_2	P_4	12.5	0.0151	S	P_2	P_5	0	0.0004	S
P_2	P_6	2	0.0007	S	P_3	P_4	24.5	0.1706	I
P_3	P_5	2.5	0.0009	S	P_3	P_6	6.5	0.0030	S
P_4	P_5	12.5	0.0151	S	P_4	P_6	16.5	0.0375	S
P_5	P_6	0	1.0000	I					

$\mathbf{P_A}, \mathbf{P_B}$=Participants of each pair

Significance level=0.0500, **RD**=Overall Rating Difference for ExcClipse, **S**=Significant, **I**=Insignificant

Table 7.8: Observed Responses and Feedback from Participants

Question	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	$\mathbf{P_5}$	$\mathbf{P_6}$
Prefer recommended query to error message from stack trace	●	●	◓	-	-	●
Query formulation frequency (Traditional)	3.5	3	-	1.5	1	2
Query formulation frequency (ExcClipse)	1	1	2.5	1	1	1
Pages browsed for a solution (Traditional)	2.5	1	-	3.5	3	2.5
Pages browsed for a solution (ExcClipse)	2.5	1	1.5	1.5	2.5	2.5
Rank of a solution page (Traditional)	0-5	0-5	-	0-8	5-10	0-5
Rank of a solution page (ExcClipse)	0-5	0-5	0-5	0-5	0-8	0-5
Context-switching between IDE and browser is problematic	●	◓	●	●	●	◓
Prefer relevant section to whole page for relevance checking	●	●	●	●	●	●
Comfortable with custom look and feel or layout	◓	●	◕	●	◓	◓
Searching within IDE is comfortable	●	●	●	●	●	●
Use web search engine for code example search	●	●	-	●	●	●
Relevant code example found (Traditional)	○	◓	-	○	◓	○
Code examples browsed to get a relevant one (ExcClipse)	1.5	2	2	1.5	3	2

$\mathbf{P_i}$=Observations for i^{th} participant

●=Totally agreed, ◕=Almost agreed, ◓=Partially agreed, ○=Not agreed

of a solution for the programming problem, and *retrieval of relevant code examples* for exception handling. In case of dichotomous ones, the recorded data reveal several interesting facts that show the potential of our proposed prototype. For example, (1) each of the participants consider that search within the IDE is convenient and helpful for problem solving, and (2) they especially prefer the supports of ExcClipse for query formulation and post-search content analysis to traditional means. Although the participants have a few minor concerns about the way the relevant section of a page is displayed, they looked pretty convinced by the overall look and feel of ExcClipse.

7.4.4 Qualitative Suggestions from Participants

We collect qualitative suggestions from the participants about ExcClipse during study sessions. This section enlists some of them as follows:

Post-search Content Analysis

- The relevant section should be hyper-linked to the corresponding section in the original web page, and the section should be highlighted for easier identification and manual analysis.
- At present, once a user requests for relevant content, ExcClipse instantly downloads the content of a page and processes the request. One participant suggests that this pulling of relevant section should be faster and that can be achieved through parallel processing (e.g., Java threading).

Code Search and Preview

- The result panel of code search should be more similar to that of web search in order to preserve the consistency in the look and feel.
- Code search query tokens should be highlighted in the code preview panel so that one can manually check the relevance of a selected example by putting less effort.

7.5 Threats to Validity

We identify a few issues with our user study worthy of discussion. First, the number of participants involved in the study is not enough, and it is a significant threat to the study. Given the length of each study session (i.e., about 1-1.5 hour) and the breadth of the study (i.e., task involved), we choose to restrict the study with six participants. More importantly, those participants (i.e., graduate students) are the potential users for our system, and three of them also have prior software development experiences in the industry. Thus we believe that although the participant size is small, they might be enough to contrast our prototype to traditional means and to explore the potential of the prototype.

Second, the participants are chosen from among the peers for our user study, and some of them are also familiar to some extent with the conducted studies in this thesis. Thus one can argue about the potential bias in the ratings by the participants for our system. While we cannot rule out the possibility of such bias, we adopted a careful technique in order to mitigate such bias in the evaluation. The study sessions with each of the participants were conducted in isolation and the evaluation was based on their instant working experience with our system as well as their best judgement. More importantly, we also analyze the problem solving experience (i.e., observation checklist) of the participants in order to determine the consistency between their experience and their ratings, and we found them quite consistent.

Third, observing problem solving practices of the participants and recording them simultaneously during execution phase are both time-consuming and error-prone. One can reduce errors in recording and analyzing observed data through screen-recording. However, this technique is more time-consuming and intrusive in nature, and participants may feel uncomfortable with recording during their work. The findings from the observed data are based on our careful observation and in-depth analysis. In order to mitigate the threat, we average the numerical observations for each of the test cases for each participant. In case of dichotomous ones, we also carefully analyze the recorded data and occasionally consult the qualitative comments from the participants to report the findings.

7.6 Summary

To summarize, in this chapter, we discussed the details and the findings of our conducted user study involving six participants. We first focus on our developed plugin prototype– *ExcClipse* (that integrates all the approaches proposed in this thesis), and discuss its different technical supports for problem solving. We then discuss the detailed design of the user study including tasks, participants, and data collection techniques– observation checklists and questionnaire. We also describe how each study session is organized and run. We then apply a list of tools and techniques to analyze the study data and to contrast ExcClipse with traditional counterpart from different perspectives. According to the findings from the study, ExcClipse is found highly promising by the participants, and it has great potential for the automated support with programming errors and exceptions. We also collect valuable suggestions from the participants for future work, and report a few threats to the validity of our study.

Chapter 8

Conclusion

8.1 Concluding Remarks

Programming errors and exceptions are inherent during development, maintenance or evolution of any software system. Software developers often look into web for working solutions or any relevant information in order to solve the encountered programming errors and exceptions in the IDE. While collecting information using traditional web search, they face several practical challenges. First, the search engines do not consider detailed context of the programming problems during search unless the developers prepare queries good enough by analyzing the context in the IDE, which is not an easy task. Thus the traditional search engines return a list of result pages which might not be properly applicable or even may be irrelevant to the problems. Second, neither stack trace nor context code of an encountered exception can be used directly as a search query because of their length. Third, manual analysis of the whole web page for relevance with a programming problem (and its context) is generally hard, and it requires a significant amount of time and effort. Another challenge that is faced by the developers is– neither traditional web search nor traditional code search is helpful in collecting readily available code examples which can be used for exception handling.

A number of existing studies [32, 33, 43, 50, 51, 55, 68, 69, 79, 80] are conducted to address such challenges above with traditional web search and code search. However, most of the approaches ignore the technical details of a programming problem (e.g., an error or an exception) and the context code (i.e., a segment in the code that triggers the exception) in the IDE, and a few of them use such information either in a limited fashion or use them in ineffective ways. Thus those approaches are either not properly applicable or not much effective for our research problems. In this thesis, we attempt to address the above four challenges using context-aware and IDE-based approaches. The approaches use technical details and context of a programming problem (e.g., an exception) effectively and in diversified ways, and provide meaningful supports with different search related activities (e.g., web search, code search). We conduct four separate studies (Chapter 3, Chapter 4, Chapter 5 and Chapter 6) followed by a user study (Chapter 7), and we have the following outcomes:

(a) The first study (Chapter 3) proposes and evaluates a context-aware meta search engine for programming errors and exceptions in the IDE. The study shows that a context-aware (i.e., considers detailed context of a programming problem) search is *more effective* than any other search approaches based on keyword matching. It also shows that our proposed meta search engine is *more likely* to return solution pages

for an exception than any of the relevant existing approaches in the literature or traditional search engines.

(b) The second study (Chapter 4) proposes and evaluates a novel search query recommender that analyzes technical details and context of an encountered error or exception, and recommends a list of suitable queries for web search. Extensive experiments and validations show that the recommended queries are relatively *more effective* than the traditional ones or the ones generated by relevant existing approaches from the literature. The study also shows that the queries from the proposed recommender are *comparable* to the expert queries collected from a user study.

(c) The third study (Chapter 5) proposes and evaluates a novel content recommendation approach that analyzes technical details and context of an encountered exception in the IDE and returns not only a noise-free version but also the most relevant section(s) of a web page. Extensive experiments followed by a limited user study show that the approach has *enough potential* to help one in post-search content analysis and in solving the exception with reduced cognitive efforts.

(d) The fourth study (Chapter 6) proposes and evaluates a code search recommender that analyzes the code under development in the IDE and recommends a list of relevant code examples from GitHub repositories for exception handling. The study shows that the existing code search approaches are either non-applicable or not much effective for such recommendation, and our recommender performs *significantly better* than a closely related existing approach.

(e) Finally, the user study (Chapter 7) involving six participants explores the potential of our proposed approaches, and contrasts our developed plugin prototype– ExcClipse to traditional search engines. The study reports that ExcClipse is *significantly preferred* to traditional means by the participants for different web search and code search related activities in the IDE.

As the findings above suggest, context-aware approaches are more effective for various recommendations (e.g., search query, web page, relevant page section and code example) than the ones that do not take context of a programming problem into consideration. We also learn that the context-aware approaches are significantly preferred by the participants in the user study than the traditional means for different search related activities. Thus we believe that our proposed context-aware approaches in all four studies have the potential to support developers in programming problem solving that involves web search and/or code search.

8.2 Future Work

While in this thesis, we deal with different conventional items such as stack trace and context code for various recommendations, in future, we plan for more granular and customized analysis for recommendation. This section discusses our future plans with the research work in the thesis.

Context-Aware Search: In this thesis, we adopt a few heuristic-based techniques in order to extract context of a programming problem discussed in a web page. The techniques are mostly effective except in the case of poorly designed web pages (i.e., pages that do not follow the well-known conventions in embedding code related items). In future, we plan to apply a more systematic approach such as *DOM tree based analysis* for such web pages for context extraction. We also plan to perform more in-depth analysis such as *topic modeling*[1] or other semantic analysis in order to determine the relevance of a page against a target programming problem.

Search Query Recommendation: In this thesis, we analyze technical details and context of a programming exception and recommend a list of suitable queries for web search. In future, we plan to extend this support to the search for other programming problems. While the current work analyzes context code and stack trace for query recommendation, in future, we plan to analyze more granular items such as code comments, identifier names in order to recommend more customized and semantically relevant queries for different programming problems.

Post-search Content Analysis: In this thesis, we extract and recommend the relevant section from a web page that is most likely to contain a solution for a programming exception at hand. However, sometimes, a relevant section could also be large itself. In future, we plan to pinpoint the possible solution location in the relevant section by highlighting different items of interest such as *target error or exception message* or *search query terms* (i.e., as suggested by the participants from the user study) and by natural language processing involving semantic analysis.

Context-Aware Code Search for Exception Handling: At present, our proposed approach recommends a list of relevant code examples containing high quality handlers for an exception of interest. However, exception handling is a frequently misunderstood concept by the developers and is greatly subjected to the context (i.e., abstraction, layer) of an application. In future, we plan to provide more directed support such as (1) whether a recommended handler code is actually applicable to the current context in the IDE or not, and (2) whether a caught exception should be handled or thrown in the current context of coding.

[1] http://en.wikipedia.org/wiki/Topic_model

Bibliography

[1] Alexa Page Rank API. URL http://data.alexa.com/data?cli=10&url=domain.name.

[2] Reading from and Writing to a Socket. URL http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html.

[3] Cosine Similarity. URL http://en.wikipedia.org/wiki/Cosine_similarity.

[4] Content Suggest Dataset. URL http://www.usask.ca/~mor543/contentsuggest/data.

[5] ContentSuggest Eclipse Plugin. URL https://marketplace.eclipse.org/content/contentsuggest.

[6] Document Object Model (DOM). URL http://en.wikipedia.org/wiki/Document_Object_Model.

[7] GitHub Code Search. URL http://developer.github.com/v3/search/.

[8] Graph Matching. URL http://en.wikipedia.org/wiki/Matching_(graph_theory).

[9] URL https://www.uni-koblenz-landau.de/campus-koblenz/fb4/west/staff/Gottron/sw-dist.

[10] Google Search Query Limit. URL http://www.google.com/support/enterprise/static/gsa/docs/admin/72/gsa_doc_set/xml_reference/request_format.html#1078040.

[11] Exception Handling Principles. URL http://howtodoinjava.com/2013/04/04/java-exception-handling-best-practices.

[12] Best Practices for Exception Handling. URL https://www.ibm.com/developerworks/library/j-ejbexcept.

[13] Java - how to instantiate inner class with reflection? URL http://stackoverflow.com/questions/17485297/java-how-to-instantiate-inner-class-with-reflection.

[14] Javaparser-Java 1.5 Parser and AST. URL http://code.google.com/p/javaparser.

[15] JSoup: Java HTML Parser. URL http://jsoup.org.

[16] Krugle Code Search Engine. URL http://krugle.com/.

[17] Logistic Regression. URL http://en.wikipedia.org/wiki/Logistic_regression.

[18] Ohloh Code Search Engine. URL http://code.ohloh.net/.

[19] Google PageRank Algorithm. URL http://pr.efactory.de/e-pagerank-algorithm.shtml.

[20] Pastebin. URL http://pastebin.com.

[21] Information Retrieval Performance Metrics. URL http://en.wikipedia.org/wiki/Information_retrieval.

[22] QueryClipse Experiment Data. URL http://www.usask.ca/~mor543/qclipse/qcdata.

[23] QueryClipse Plugin for Eclipse. URL http://www.usask.ca/~mor543/tools/tools.php.

[24] SurfClipse Experiment Data. URL http://homepage.usask.ca/~mor543/sc/info.

[25] SurfClipse Plug-in. URL https://marketplace.eclipse.org/content/surfclipse.

[26] SurfExample Portal. URL http://www.usask.ca/~mor543/surfexample.

[27] java.io.EOFException using readObject. URL http://www.coderanch.com/t/522094/java-io/java/java-io-EOFException-readObject.

[28] Readability Library. URL http://www.arrestedcomputing.com/readability.

[29] Weka. URL http://www.cs.waikato.ac.nz/ml/weka/.

[30] SurfClipse Web Service. URL https://srlabg53-2.usask.ca/wssurfclipse.

[31] A. Arif, M.M. Rahman, and S.Y. Mukta. Information Retrieval by Modified Term Weighting Method Using Random Walk Model with Query Term Position Ranking. In *Proc. ICSPS*, pages 526–530, 2009.

[32] S. Bajracharya, T. Ngo, E. Linstead, P. Rigo, Y. Dou, P. Baldi, and C. Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Proc. OOPSLA*, pages 25–26, 2006.

[33] E. A. Barbosa, A. Garcia, and M. Mezini. Heuristic Strategies for Recommendation of Exception Handling Code. In *Proc. SBES*, pages 171–180, 2012.

[34] R. Blanco and C. Lioma. Random walk term weighting for information retrieval. In *Proc. SIGIR*, pages 829–830, 2007.

[35] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.

[36] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proc. SIGCHI*, pages 513–522, 2010.

[37] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *TSE*, 36(4):546–558, 2010.

[38] B. Cabral and P. Marques. Exception Handling: A Field Study in Java and .NET. In *Proc. ECOOP*, pages 151–175, 2007.

[39] M.J. Cafarella. *Extracting and Managing Structured Web Data*. PhD thesis, 2009.

[40] D. Cai, S. Yu, J. Wen, and W. Ma. Extracting Content Structure for Web Pages Based on Visual Representation. In *Proc. APWeb*, pages 406–417, 2003.

[41] B. M. Chang, J. W. Jo, K. Yi, and K. M. Choe. Interprocedural Exception Analysis for Java. In *Proc. SAC*, pages 620–625, 2001.

[42] Y. Chun, L. Yazhou, and Q. Qiong. An Approach for News Web-Pages Content Extraction Using Densitometric Features. In *Advances in Electric and Electronics*, volume 155, pages 135–139. 2012.

[43] J. Cordeiro, B. Antunes, and P. Gomes. Context-based Recommendation to Support Problem Solving in Software Development. In *Proc. RSSE*, pages 85 –89, June 2012.

[44] B.D. Davison. Recognizing Nepotistic Links on the Web. In *Proc. AAAI*, pages 23–28, 2000.

[45] F. Ensan, E. Bagheri, and M. Kahani. The application of users' collective experience for crafting suitable search engine query recommendations. In *Proc. CNSR*, pages 148–156, 2007.

[46] T. Furche, G. Gottlob, G. Grasso, G. Orsi, C. Schallhart, and C. Wang. Little Knowledge Rules the Web: Domain-centric Result Page Extraction. In *Proc. RR*, pages 61–76, 2011.

[47] 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.

[48] D. Gibson, K. Punera, and A. Tomkins. The Volume and Evolution of Web Page Templates. In *Proc. WWW*, pages 830–839, 2005.

[49] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Commun. ACM*, 18(12): 683–696, 1975.

[50] T. Gottron. Content Code Blurring: A New Approach to Content Extraction. In *Proc. DEXA*, pages 29–33, 2008.

[51] Thomas Gottron. Combining Content Extraction Heuristics: The CombinE System. In *Proc. IIWAS*, pages 591–595, 2008.

[52] Z. Gu, E.T. Barr, D. Schleck, and Z. Su. Reusing debugging knowledge via trace-based bug search. In *Proc. OOPSLA*, pages 927–942, 2012.

[53] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proc. ICSE*, pages 223–226, 2010.

[54] B. Hartmann, D. MacDougall, J. Brandt, and S.R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proc. CHI*, pages 1019–1028, 2010.

[55] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*, pages 117–125, 2005.

[56] R. Islam, B.D. Sarker, and R. Islam. An effective term weighting method using random walk model for information retrieval. In *ICCCE*, pages 1357–1362, 2008.

[57] M. Kim, Y. Kim, W. Song, and A. Khil. Main Content Extraction from Web Documents Using Text Block Context. In *Proc. DEXA*, pages 81–93. 2013.

[58] C. Kohlschutter, P. Fankhauser, and W. Nejdl. Boilerplate Detection Using Shallow Text Features. In *Proc. WSDM*, pages 441–450, 2010.

[59] 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.

[60] N. Kushmerick. Learning to Remove Internet Advertisements. In *Proc. AGENTS*, pages 175–181, 1999.

[61] C. Le Goues and W. Weimer. Measuring Code Quality to Improve Specification Mining. *TSE*, 38(1): 175–190, 2012.

[62] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: On novices' interactions with error messages. In *Proc. ONWARD*, pages 3–18, 2011.

[63] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a Method Co-change Pattern to Identify Highly Coupled Methods: An Empirical Study. In *Proc. ICPC*, pages 103–112, 2013.

[64] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic Identification of Important Clones for Refactoring and Tracking. In *Proc. SCAM*, page 10, 2014 (to appear).

[65] S.M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What Makes a Good Code Example?: A Study of Programming Q & A in StackOverflow. In *Proc. ICSM*, pages 25–34, 2012.

[66] A. Nenkova, R. Passonneau, and K. McKeown. The pyramid method: Incorporating human content selection variation in summarization evaluation. *ACM Trans. Speech Lang. Process.*, 4(2), 2007.

[67] 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.

[68] D. Pinto, M. Branstein, R. Coleman, W. B. Croft, M. King, W. Li, and X. Wei. QuASM: A System for Question Answering Using Semi-structured Data. In *Proc. JCDL*, pages 46–55, 2002.

[69] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Proc. CSMR*, pages 57–66, 2013.

[70] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.

[71] D. Poshyvanyk, M. Petrenko, and A. Marcus. Integrating COTS Search Engines into Eclipse: Google Desktop Case Study. In *Proc. IWICSS*, pages 6–, 2007.

[72] 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.

[73] M.M. Rahman, S. Yeasmin, and C.K. Roy. An IDE-Based Context-Aware Meta Search Engine. In *Proc. WCRE*, pages 467–471, 2013.

[74] 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.

[75] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, pages 172–181, 2008.

[76] N. Sawadsky, G.C. Murphy, and R. Jiresal. Reverb: Recommending code-related web pages. In *Proc. ICSE*, pages 812–821, 2013.

[77] H. Shah, C. Görg, and M. J. Harrold. Visualization of Exception Handling Constructs to Support Program Understanding. In *Proc. SoftVis*, pages 19–28, 2008.

[78] Z. Soh, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. Towards understanding how developers spend their effort during maintenance activities. In *Proc. WCRE*, pages 152–161, 2013.

[79] F. Sun, D. Song, and L. Liao. DOM Based Content Extraction via Text Density. In *Proc. SIGIR*, pages 245–254, 2011.

[80] W. Takuya and H. Masuhara. A Spontaneous Code Recommendation Tool Based on Associative Search. In *Proc. SUITE*, pages 17–20, 2011.

[81] 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.

[82] H. Vahabi, M. Ackerman, D. Loker, R. Baeza-Yates, and A. Lopez-Ortiz. Orthogonal query recommendation. In *Proc. RecSys*, pages 33–40, 2013.

[83] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Genealogical Insights into the Facts and Fictions of Clone Removal. *ACR*, 13(4):30–42, 2013.

Appendix A

User Study Test Cases

A.1 Exception Test Case 1 (EC_1)

A.1.1 Stack Trace

```
java.io.EOFException
at java.io.ObjectInputStream$PeekInputStream.readFully(Unknown Source)
at java.io.ObjectInputStream$BlockDataInputStream.readShort(Unknown Source)
at java.io.ObjectInputStream.readStreamHeader(Unknown Source)
at java.io.ObjectInputStream.<init>(Unknown Source)
at cores.TestCase1.main(TestCase1.java:15)
```

A.1.2 Context Code

```
12 try{
13         File file=new File("./data/recordlist.dat");
14          FileInputStream fis = new FileInputStream(file);
15          ObjectInputStream ois = new ObjectInputStream(fis);
16          ArrayList<Record> currentList = new ArrayList<Record>();
17          //restore the number of objects
18          int size = ois.readInt();
19          for (int i=0; i<size; i++) {
20              Record current = (Record) ois.readObject();
21              currentList.add(current);
22          }
23   }catch(Exception e){
24     e.printStackTrace();
25   }
```

A.2 Exception Test Case 2 (EC_2)

A.2.1 Stack Trace

```
javax.imageio.IIOException:  Not a JPEG file:  starts with 0xff 0xd9
at com.sun.imageio.plugins.jpeg.JPEGImageReader.readImageHeader(Native Method)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.readNativeHeader(Unknown Source)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.checkTablesOnly(Unknown Source)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.gotoImage(Unknown Source)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.readHeader(Unknown Source)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.readInternal(Unknown Source)
at com.sun.imageio.plugins.jpeg.JPEGImageReader.read(Unknown Source)
at javax.imageio.ImageReader.read(Unknown Source)
at cores.TestCase2.main(TestCase2.java:19)
```

A.2.2 Context Code

```
13 try{
14     File imgFile=new File("./data/myimg.jpg");
15     Iterator readers = ImageIO.getImageReadersByFormatName("jpg");
16     ImageReader reader = (ImageReader) readers.next();
```

```
17      ImageInputStream iis = ImageIO.createImageInputStream(imgFile);
18      reader.setInput(iis, true, true);
19      BufferedImage image = reader.read(0);
20      }catch(Exception exc){
21        exc.printStackTrace();
22      }
```

A.3 Exception Test Case 3 (EC_3)

A.3.1 Stack Trace

```
java.lang.IllegalAccessException: Class cores.TestCase5 can not access a
    member of class java.util.HashMap$HashIterator with modifiers "public final"
at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(Unknown Source)
at java.lang.reflect.AccessibleObject.checkAccess(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
at cores.TestCase5.main(TestCase5.java:19)
```

A.3.2 Context Code

```
13  try{
14        Dummy dummy=new Dummy();
15        Set<String> myStr = new HashSet<String>();
16        myStr.add(dummy.toString());
17        Iterator itr = myStr.iterator();
18        Method mtd = itr.getClass().getMethod("hasNext");
19        System.out.println(mtd.invoke(itr, null));
20        }catch(Exception exc){
21          exc.printStackTrace();
22        }
```

A.4 Exception Test Case 4 (EC_4)

A.4.1 Stack Trace

```
java.net.UnknownServiceException: protocol does not support output
at java.net.URLConnection.getOutputStream(Unknown Source)
at cores.TestCase6.main(TestCase6.java:14)
```

A.4.2 Context Code

```
10  try{
11        URL url = new File("./data/filedata.txt").toURL();
12        URLConnection connection = url.openConnection();
13        connection.setDoOutput(true);
14        OutputStream output = connection.getOutputStream();
15        System.out.println(output);
16      }catch(Exception exc){
17        exc.printStackTrace();
18      }
```

Appendix B

User Study Data Collection Techniques

B.1 Observation Checklist

B.1.1 Observation Checklist for Traditional Search

We make the following observations when a participant performs the study tasks using a traditional search engine (e.g., Google) or a traditional code search engine (e.g., GitHub):

Query Formulation

- Does the participant analyze the stack trace or the context code of the encountered exception? [Yes/No]
- Does he or she solve the exception without web search? [Yes/No]
- Does he or she use only the error message as a search query? [Yes/No]
- Does he or she use other tokens from the stack trace in the search query? [Yes/No]
- Does he or she use Google/Bing/Yahoo queries for search? [Yes/No]

Web Page Search

- Which search engine does he or she use? [Google, Bing, Yahoo]
- Does he or she try multiple search engines? [Yes/No]
- How many web pages does he or she browse to get a solution?
- Rank(s) of the solution(s) in the result list? [0-5, 5-10, 10-15]
- How many attempts (i.e., query reformulation) does he or she make for a single search task?

Post-search Content Analysis

- Does he or she check page title for browsing? [Yes/No]
- Does he or she check the page description (i.e., meta description by search engines) for browsing? [Yes/No]
- Does he or she check an entire web page for relevant content sections? [Yes/No]
- Does he or she move back and forth between IDE and browser for relevance checking of a page? [Yes/No]
- Is the moving back and forth inconvenient for him or her? [Yes/No]

Code Example Search

- Does he or she use IDE provided features for exception handling? [Yes/No]
- Does he or she analyze the context code to prepare a search query for code search? [Yes/No]
- Is he or she able to formulate a search query including exception name and tokens? [Yes/No]
- Does he or she use a code search engine for code example search? [Yes/No]
- Does he or she get any query support from the traditional code search engine (GitHub or Krugle)? [Yes/No]

- How many results does he or she browse to get a relevant example for exception handling?
- How many attempts (i.e., query reformulation) does he or she make for a single task?

B.1.2 Observation Checklist for ExcClipse

We make the following observations when a participant performs the study tasks using our proposed prototype–*ExcClipse*:

Query Formulation

- Does the participant use a search query from the recommended list? [Yes/No]
- Does he or she modify the query and develop a custom query for search? [Yes/No]
- Which ranked query does provide the solution mostly?
- Does he or she prefer a recommended search query over the error message from stack trace? [Yes/No]

Web Page Search

- How many web pages does he or she browse to get a solution?
- Rank(s) of the solution(s) in the result list? [0-5, 5-10, 10-15]
- How many attempts (i.e., query reformulation) does he or she make for a single search task?
- Does he or she feel comfortable with web search within the IDE? [Yes/No]
- Does he or she consider switching context between IDE and web browser problematic? [Yes/No]

Post-search Content Analysis

- Are page title and page description enough for relevance checking of a web page? [Yes/No]
- Is a relevant section from the page better option for relevance checking of the page? [Yes/No]
- Is it (i.e., consulting relevant section) more effective than consulting the entire web page? [Yes/No]
- He or she does not need to move back and forth for relevance checking, is it convenient? [Yes/No]
- Is he or she comfortable with the custom layout of relevant section panel? [Yes/No]

Code Example Search

- Does he or she use a recommended query for code search? [Yes/No]
- Does he or she need to reformulate the query for search? [Yes/No]
- How many results does he or she browse to get a relevant example for exception handling?
- How many attempts does he or she make for a single task?
- Is code example previewing a good choice before actually working with an example? [Yes/No]
- Does the code token highlighting feature help to select an appropriate example? [Yes/No]

B.2 Questionnaire

We ask the following 15 questions in the form of questionnaire to each participant once they complete the four search-related tasks:

- Is *switching context* between IDE and web browser inconvenient or distracting for your work?

  Options: (a) Very problematic (b) Problematic (c) Does not matter (d) Helpful for work (e) Very helpful for work

- *Preparing a search query* for a programming error or an exception is generally

  Options: (a) Very hard (b) Hard (c) Neither hard nor easy (d) Easy (e) Very easy

- How would you evaluate an approach that fetches top-ranked results from multiple search engines such as Google, Bing and Yahoo all at once for the same query and then merges the results with reliable ranking?

  Options: (a) Excellent (b) Good (c) Somewhat good (d) Neither good nor bad (e) Not good

- What do you think about the search queries *recommended by Google/Bing/Yahoo search engines* when you start typing?

  Options: (a) Very relevant for my exception (b) Relevant to my exception (c) Sometimes relevant (d) Hardly relevant (e) Not relevant at all

- *Checking relevance* of a web page in the browser with the exception in the IDE is

  Options: (a) Very hard (b) Hard (c) Neither hard nor easy (d) Easy (e) Very easy

- The *code examples by ExcClipse for exception handling* are

  Options: (a) Very helpful (b) Helpful (c) Not sure (d) Less helpful (e) Least helpful

  [1=least useful or least effective, 10=most useful or most effective]

- Rate the *Search query formulation support* for programming errors and exceptions between 1-10 (least to most). *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *accuracy and effectiveness* (i.e., the returned results are actually useful) of search result ranking between 1 to 10 (least to most). *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *support for post-search content analysis* (e.g., how easily can you locate a relevant page or relevant content from the page?) between 1 to 10 (least to most).

  Options: (a) Traditional — (b) ExcClipse —

- Rate the *query formulation support* (i.e., quality and ranking of query) for code example search between 1 to 10 (least to most). *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *relevance and accuracy* of code example search feature between 1 to 10 (least to most). *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *usability* (i.e., easy to use) between 1 and 10 (least to most).

  Options: (a) Traditional — (b) ExcClipse —

- Rate the *efficiency* (i.e., consumes less effort or time but provides more gain) of the search providers for problem solving between 1 and 10 (least to most) *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *overall look and feel* (e.g., result relevance visualization and relevant content visualization) of the web search providers between 1 and 10 (least to most) *Options:* (a) Traditional — (b) ExcClipse —

- Rate the *overall look and feel* (e.g., result relevance visualization and code example preview) of the code search providers between 1 and 10 (least to most) *Options:* (a) Traditional — (b) ExcClipse —