AN ASSISTANT FRAMEWORK FOR PROCURING COMPONENTS AND CODE SAMPLES

BY

NAIYANA TANSALARAK
B.S. CHIANGMAI UNIVERSITY, THAILAND
M.S. UNIVERSITY OF MASSACHUSETTS LOWELL

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF SCIENCE IN COMPUTER SCIENCE UNIVERSITY OF MASSACHUSETTS LOWELL

Signature of Author: ___________________________ Date: ________________

Signature of Dissertation Supervisor:

Dr. Kajal T. Claypool ___________________________

Signature of Other Dissertation Committee Members:

Dr. Robert Lechner ___________________________
Dr. George Heineman ___________________________
AN ASSISTANT FRAMEWORK FOR PROCURING COMPONENTS AND CODE SAMPLES

BY

NAIYANA TANSALARAK

ABSTRACT OF A DISSERTATION SUBMITTED TO THE FACULTY OF THE DEPARTMENT OF COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF SCIENCE IN COMPUTER SCIENCE UNIVERSITY OF MASSACHUSETTS LOWELL 2006

Dissertation Supervisor: Dr. Kajal T. Claypool
Assistant Professor, Department of Computer Science
ABSTRACT

Software reuse has been targeted as a panacea for reducing software costs and for improving both software quality and programmer productivity. Today, it is accepted practice for software developers to reuse software units, that range from ASCII units such as code fragments, to binary units such as functions, modules and even entire application systems, to both guide and reduce their development efforts. These software units are typically bundled with library or framework packages, are available as in-house and off-the-shelf components, or are accessible via books, manuals, web sites, live application code and code that the developer has written previously.

Software unit procurement is the first key step of the reuse process at the software implementation stage, and is often broken down into component procurement and code sample procurement based on the two popular types of software units. While tools exist to assist developers in this procurement process, the tools are limited in their capabilities by the type of queries they provide, the efficient matching/mining techniques that underlie the procurement process and the effective ranking techniques used to evaluate the procured units. We broadly classify these limitations as quality of procured units and the effective ranking of the same.

This dissertation contributes solutions that address the quality and ranking of retrieved software units. As part of this dissertation, we have developed two assistant frameworks, the XML-based Component Discovery and Evaluation (XCoDE) framework and the XML-based Code Snippet Mining and Ranking (XSnippet) framework for procuring components and code samples, respectively.

The XCoDE framework allows developers to query for components based on the component information. Our approach is based on a unique Quality of Match (QoM)
metric that qualitatively and quantitatively evaluates the match of two given components. The QoM metric forms the basis for a disciplined application of different matching techniques to all aspects of a component, thereby exploiting the diversity of syntactic and semantic information inherent in a component. Our experiments have shown that XCoDE has significant potential to provide higher accuracy than individual matching techniques and other combination techniques.

The XSnippet framework allows developers to query for code samples based on the context of code under development. Our approach is based on a novel graph-based code mining algorithm that supports the range of object instantiation-specific queries, and an innovative context-sensitive ranking heuristic that has been experimentally proven to provide better ranking of best-fit code samples than context-independent heuristics. Our experimental evaluation has shown that XSnippet has significant potential to assist developers, and provides better coverage of tasks and better rankings for best-fit code samples than other code assistant systems.
ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr.Kajal T. Claypool, for her gracious help and encouragement during my Sc.D. tenure. She taught me how to approach research problems, express my ideas and finally how to accomplish my goals. She also helped me complete the writing of this dissertation. I also thank my committee members, Dr.Robert Lechner and Dr.George Heineman, for taking time to read my dissertation and for providing insightful suggestions.

A special thanks goes to Dr.Kanchit Malaiwong, Dr.Kwan Sitathani, MR Nongkran Chompoonut and others at National Electronics and Computer Technology Center, National Science and Technology Development Agency, Ministry of Science THAILAND for giving me an opportunity to pursue my Sc.D. in Computer Science. I also thank the Royal Thai Government for providing financial support as well as Computer Science Department at UMass-Lowell for providing financial support, personal office and equipment.

I am also grateful for the support I received from everyone in the Computer Science Department at UMass-Lowell, particularly the students who volunteered their time for my experiments. I would like to extend my thanks to faculty members and students in Database and Software Lab (DSL) for providing a venue for exchanging research ideas that helped my research practices and skills. I also thanks all my friends both in THAILAND and USA for encouraging and believing in me.

Last, but not least, I thank my family: my parents, Mr.Niwat and Mrs.Pensri Tansalarak, for giving me my life, for educating me with aspects from both arts and sciences, and for unconditional support and encouragement to pursue my interests and especially in helping reduce my stress during my Sc.D program. My
sister, Ms.Phattrra Tansalarak, for taking good care of our parents while I was
away from home and for believing in me. My brothers, Mr.Thanyawat and Nut-
thawut Tansalarak, and my sister-in-laws, Ms.Duangduen, Supakiti and Suppakorn
Sahavechaphan, for always asking when I was going to graduate and be back home.
Finally, a million thanks to my beloved husband, Mr.Krissada Sahavechaphan, for
his support and understanding when I needed to focus on my work and for always
listening to my complaints and frustrations. Without him, I would not have been
able to work on my degree.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xvii</td>
</tr>
<tr>
<td>I Introduction</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Issues in Software Unit Procurement</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Component Procurement</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Code Sample Procurement</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Our Work</td>
<td>10</td>
</tr>
<tr>
<td>1.3.1 The XCoDE Framework</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2 The XSnippet Framework</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Organization of Dissertation</td>
<td>14</td>
</tr>
<tr>
<td>II XCoDE: An XML-based Component Discovery and Evaluation Framework</td>
<td>16</td>
</tr>
<tr>
<td>2. XCM: THE COMPONENT ONTOLOGY CONCEPTS</td>
<td>19</td>
</tr>
<tr>
<td>3. PRIMITIVE MATCH</td>
<td>23</td>
</tr>
</tbody>
</table>
3.1 Label Match ................................................................. 24
  3.1.1 Qualitative and Quantitative Analysis ....................... 24
  3.1.2 Label Match Algorithm ................................. 25
3.2 Type Match ................................................................. 26
  3.2.1 Qualitative and Quantitative Analysis ....................... 26
3.3 Dewey-based Type ...................................................... 30
  3.3.1 Type Match Algorithm ................................. 34

4. QOM - QUALITY OF MATCH METRIC ................................. 37
  4.1 The Qualitative Analysis - Match Taxonomy ................. 37
    4.1.1 Micro Match ............................................. 38
    4.1.2 Macro Match ......................................... 39
  4.2 The Quantitative Analysis - Weight-Based Match Model .... 40
    4.2.1 Micro Match Model ..................................... 41
    4.2.2 Macro Match Model ................................... 42

5. THE QOMYM ALGORITHM ............................................... 44

6. EXPERIMENTAL RESULTS ............................................... 47
  6.1 Experimental Setup and Methodology ......................... 47
  6.2 Determining Threshold and Significance Values ............. 49
  6.3 QoMym Match Quality ........................................... 52

7. RELATED WORK ........................................................... 56
  7.1 Match Algorithms .................................................. 56
    7.1.1 Component Match Algorithms .......................... 56
    7.1.2 Schema Match Algorithms ................................ 61
  7.2 Component Repository ............................................. 63
  7.3 Evaluation Techniques ............................................ 65

8. SUMMARY ................................................................. 69
III  **XSnippet**: An XML-based Snippet Mining and Ranking Framework  

9. THE FRAMEWORK OVERVIEW OF XSNIPPET ................................. 73  

10. PROGRAM CONTEXT AND QUERIES ........................................... 76  

   10.1 Types of Programming Context ............................................. 76  

   10.2 Snippet Queries .............................................................. 78  

11. THE SOURCE CODE MODEL ...................................................... 83  

   11.1 Nodes in Source Code Model ............................................... 84  

   11.2 Edges in Source Code Model .............................................. 89  

   11.3 Algorithm: Transforming a Java Source Class $C_s$ to the Code Model Instance $CM_s$ ......................................................... 95  

   11.4 Transforming a Mining Path to the Java Code Snippet .............. 102  

12. SNIPPET MINING ................................................................. 106  

   12.1 DFSMINE Algorithm ......................................................... 107  

   12.2 Extensions to the DFSMINE Algorithm .................................. 111  

13. THE RANKING TECHNIQUE ..................................................... 120  

   13.1 Length Ranking ............................................................... 120  

   13.2 Frequency Ranking ........................................................... 122  

   13.3 Context-based Ranking ..................................................... 122  

14. EXPERIMENTAL EVALUATION ............................................... 126  

   14.1 The Experiment Setup and Methodology ................................ 129  

   14.2 Hypothesis 1. Effect of Query Types on Task Completion .......... 131  

   14.3 Hypothesis 2. Impact of Contextual Information on Ranking ...... 135  

   14.4 Hypothesis 3. Effects of Query Types on Ranking .................. 139  

   14.5 Hypothesis 4. Analysis of XSnippet on Assisting Developers ...... 142  

   14.6 Hypothesis 5. Head-to-Head Comparison ................................ 151  

      14.6.1 Prospector Versus XSnippet ......................................... 151  

      14.6.2 XSnippet Versus Prospector and Strathcona ..................... 153  

15. RELATED WORK ................................................................. 159  

   15.1 Context-sensitive Mining ................................................ 159  

   15.2 Context-independent Mining ............................................. 163
16. SUMMARY ................................................................. 165

IV Conclusions and Future Work ............................................ 167

17. CONCLUSIONS AND FUTURE WORK ............................... 168

17.1 Contributions of this Dissertation ................................. 168
17.1.1 Component Procurement ....................................... 168
17.1.2 Code Sample Procurement .................................... 170

17.2 Future Work ............................................................. 171
17.2.1 Queries ............................................................... 171
17.2.2 Virtual Repository ............................................... 172
17.2.3 Optimization of Search Space .................................. 173

APPENDICES

A. TASKS IN USER STUDY ............................................... 175

A.1 Programming Task A .................................................. 176
A.2 Programming Task B .................................................. 178
A.3 Programming Task C .................................................. 180
A.4 Programming Task D .................................................. 182

BIBLIOGRAPHY .................................................................. 184
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Reuse Process.</td>
<td>2</td>
</tr>
<tr>
<td>1.2 The Software Life Cycle with Reusable Software Artifacts.</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The Overall Retrieval and Evaluation Processes.</td>
<td>4</td>
</tr>
<tr>
<td>1.4 The Current Component Discovery and Evaluation.</td>
<td>7</td>
</tr>
<tr>
<td>2.1 The Component Structure.</td>
<td>20</td>
</tr>
<tr>
<td>2.2 The XCM Concept Schema.</td>
<td>21</td>
</tr>
<tr>
<td>2.3 The Structure of The Component JSlider.</td>
<td>22</td>
</tr>
<tr>
<td>2.4 The XCM of the JSlider component.</td>
<td>22</td>
</tr>
<tr>
<td>3.1 The Label Match Algorithm. The Method <code>computeMaxMatch</code>&lt;br&gt;(<code>match</code>) Corresponds to the Numerator of Equation 3.1.</td>
<td>27</td>
</tr>
<tr>
<td>3.2 The Partial Class Type Hierarchy along with Dewey Numbers.</td>
<td>28</td>
</tr>
<tr>
<td>3.3 The Partial Interface Type Hierarchies along with the Dewey Numbers</td>
<td>33</td>
</tr>
<tr>
<td>3.4 The <code>typeMatch</code> Algorithm.</td>
<td>35</td>
</tr>
<tr>
<td>3.5 The <code>typeMatchHelper</code> Algorithm.</td>
<td>36</td>
</tr>
<tr>
<td>4.1 The Micro Match.</td>
<td>38</td>
</tr>
<tr>
<td>4.2 The Examples of Micro Matches.</td>
<td>39</td>
</tr>
<tr>
<td>4.3 The Macro Match.</td>
<td>40</td>
</tr>
<tr>
<td>4.4 The Example of Macro Match.</td>
<td>41</td>
</tr>
</tbody>
</table>
5.1 The Overall Execution of QoMym ............................................. 44
5.2 The QoMym Algorithm ......................................................... 45
5.3 The microMatch Algorithm .................................................. 46
5.4 The macroMatch Algorithm .................................................. 46
6.1 The Overall Architecture of the QoM System ............................ 48
6.2 Variance in the precision and recall obtained for different label threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis. ............................................. 50
6.3 Variance in the precision and recall obtained for different signature threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis. ............................................. 51
6.4 Variance in the precision and recall obtained for different signature significance values as well as the micro thresholds. Label and signature thresholds were kept constant at 0.8 and 0.7 respectively. ............................................. 51
6.5 Variance in the precision and recall obtained for different macro values. Signature significance values and micro thresholds were kept constant at 0.2 and 0.7 respectively. ............................................. 52
6.6 The Query Component - Calendar ........................................... 53
6.7 The Number of Qualified Components Returned For the Different Matching Techniques ............................................. 54
6.8 The Match Quality of the Different Matching Techniques .......... 54
9.1 The framework overview of xSnippet ..................................... 74
10.1 Code Snippets that Show Six Different Ways of Instantiating an ICompilationUnit Object ............................................. 77
10.2 The JavaMetricsView Class showing the Method getCompilationUnit() ............................................. 78
10.3 Code Segment of `JavaMetricsView` Class showing the Object Instantiation of `ICompilationUnit`.......................... 81

10.4 Code Segment of `AddTraceStatementsEditorAction` Class showing Object Instantiation of `ICompilationUnit`........... 82

11.1 The Node Notations defined in the Source Code Model $CM$. ....... 84

11.2 The Mapping from Java Source Class to `type` Nodes. ............... 85

11.3 The Mapping from Java Source Class to `object` Nodes. ............... 87

11.4 The Roles of Nodes in a Source Code Model Instance $cm_s$. ......... 88

11.5 The Mapping from Java Source File to Method Nodes. ............... 89

11.6 The Edge Notations defined in the Source Code Model............... 89

11.7 The Mapping from Java Source File to Structural Edges.............. 91

11.8 The Mapping from Java Source File to Method Edges............... 92

11.9 The Mapping from Java Source File to Assignment Edges........... 93

11.10 The Mapping from Java Source File to Parameter Edge............ 94

11.11 The High-level Algorithm for Transforming a Source Class to Nodes $N_{CM}$ and Edges $E_{CM}$ in an Instance of Source Code Model $cm_s$. ........................................ 96

11.12 The Partial Source Code Model Instance of the Source Class `JavaMetricsView`........................................ 97

11.13 The Source Class `JavaMetricsView`.................................. 98

11.14 The High-level Algorithm for Transforming an Expression to Nodes $N_{CM}$ and Edges $E_{CM}$ in an Instance of Source Code Model $cm_s$ (1/2)............................................. 99

11.15 The High-level Algorithm for Transforming an Expression to Nodes $N_{CM}$ and Edges $E_{CM}$ in an Instance of Source Code Model $cm_s$ (2/2)............................................. 100

11.16 The Transformation of the Method Invocation

```
parent.setLayout(f1_1).
```

............................................. 100
11.17 The Transformation of the Sequence of Method Invocations
getViewSite().getWindowWorkbench().getSelectionService().
addSelectionListener(this). .............................................. 101

11.18 The Transformation of the Class Instance Creation new
FillLayout() ............................................................... 101

11.19 The Transformation of Cast Expression “(IJavaElement)
ss.getFirstElement(f)” .................................................. 102

11.20 The Transformation of Assignment Statement “IJavaElement je =
(IJavaElement) ss.getFirstElement(f)” .............................. 103

11.21 The Transformation of Two Nodes Connected via a Method
Edge ................................................................. 104

11.22 The Transformation of Two Nodes Connected via a Method Edge
that is Connected with a Parameter Node via a Parameter
Edge ................................................................. 104

11.23 The Transformation of Two Nodes Connected via an Assignment
Edge ................................................................. 105

12.1 The snippet mining process ........................................ 107

12.2 The DFSMINE algorithm ........................................... 108

12.3 Back Traversal for Query $I Q = (t q = ICompilationUnit, CT =
\{\})$. The code model instance of method
getCompilationUnit() in class JavaMetricsView is shown.
The example highlights the basic back traversal of the
DFSMINE algorithm ..................................................... 109

12.4 Code Snippets for Query $I Q = (t q = ICompilationUnit, CT = \{\})$
based on Code Model Instance in Figure 12.3 ..................... 111

12.5 Code Snippets for Query $I Q = (t q = ICompilationUnit, CT =
\{IJavaElement\})$ based on Code Model Instance in
Figure 12.3 ................................................................. 111

12.6 Back Traversal for Query $I Q = (t q = ICompilationUnit, CT = \{\})$.
The code model instance of method getCompilationUnit() in
class JavaMetricsView is shown. The example highlights the
back traversal of all edges ........................................... 113
12.7 Code Snippets for Query $\mathcal{I}Q = (t_q = \text{ICompilationUnit}, \mathcal{C}T = \{\})$
   based on Code Model Instance in Figure 12.6. .................. 113

12.8 The BFSMINE-EXT algorithm (1/3). ......................... 114

12.9 The BFSMINE-EXT algorithm (2/3). ......................... 115

12.10 The BFSMINE-EXT algorithm (3/3). ......................... 116

12.11 Back Traversal for Query $\mathcal{I}Q = (t_q = \text{ICompilationUnit}, \mathcal{C}T =$
   \{ISelection\}). The code model instances of methods
   getCompilationUnit() and selectionChanged() in class
   JavaMetricsView are shown. The example highlights the back
   trace across method boundaries via a parameter to the method
   invocation point. .............................................. 117

12.12 Code Snippets for Query $\mathcal{I}Q = (t_q = \text{ICompilationUnit}, \mathcal{C}T = \{\})$
   based on Code Model Instance in Figure 12.11. .............. 118

12.13 Back Traversal for Query $\mathcal{I}Q = (t_q = \text{ICompilationUnit}, \mathcal{C}T = \{\})$.
   The code model instances of methods getCompilationUnit() and
   selectionChanged() in class JavaMetricsView are shown.
   The example highlights the back trace across method boundaries
   via the invocation of a local method. ....................... 119

14.1 The Architectural Overview of the XSnippet framework. ...... 126

14.2 A Snapshot of the XSnippet Interface - Query. .............. 127

14.3 A Snapshot of the XSnippet Interface - Result. .............. 128

14.4 The Percentage of Tasks Supported by the Different Types of
   Instantiation Queries - $\mathcal{I}Q_G$, $\mathcal{I}Q_T$, and $\mathcal{I}Q_P$. .......... 132

14.5 Characteristics of the Tasks Supported by the Different Types of
   Instantiation Queries. .................................... 133

14.6 The Number of Queries Needed to Complete the Tasks using
   Different Query Types $\mathcal{I}Q_T$, $\mathcal{I}Q_P$, $\mathcal{I}Q_G$. .............. 135

14.7 Cumulative Distribution of the Best-Fit Code Snippet Ranking for
   Different Ranking Heuristics using $\mathcal{I}Q_T$. .................. 137

14.8 Cumulative Distribution of the Best-Fit Code Snippet Ranking for
   Different Ranking Heuristics using $\mathcal{I}Q_P$. ............... 138
14.9 Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using $\mathcal{I}_Q$. ......................... 139

14.10 Variation in the Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $\mathcal{I}_Q_T$. .............. 140

14.11 Variation in the Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $\mathcal{I}_Q_P$. .............. 141

14.12 Variation in the Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $\mathcal{I}_Q_G$. .............. 142

14.13 The Test Harness: The Demographic Screen. ......................... 144

14.14 The Test Harness: The Task Screen. ................................. 145

14.15 The Test Harness: The Questionnaire Screen. ........................ 146

14.16 Percentage of Participants that Completed the Given Four Tasks in the *Hint* and *No Hint* Groups. ................................. 147

14.17 Percentage of Participants that Completed the Given Four Tasks in the Two Java Experience Classes: *less than 1 year* and *greater than 1 year* in Java Experience. ................................. 148

14.18 Development Time (Individual and Average) For Completing the Four Tasks. The Results are Categorized by the *Hint* and *No Hint* Groups. ................................. 149

14.19 Average Number of Code Snippet Selections For Different Ranks of Best-fit Code Snippets. ................................. 151

14.20 Percentage of Tasks Completed By Prospectors $\mathcal{I}_Q_T^I$ and $XSnippet$ $\mathcal{I}_Q_T$ and $\mathcal{I}_Q_G$. The results are categorized by the existence of $t_b$ in the code context. ................................. 153

14.21 Percentage of Queries Supported By Prospector and $XSnippet$. ................................. 156


14.23 Percentage of Queries Supported By Strathcona and $XSnippet$. ................................. 158

15.1 The Partial Signature Graph from AST to IDocument ......................... 161
15.2 The Prospector’s Partial Snippet Results where AST and IDocument is an 
input and output type. ........................................... 162

16.1 An Overview of the XSnippet Framework................................. 166

A.1 The Partial JavaMetricsView Class. ................................. 176

A.2 The Expectation of Programming Task A. ......................... 177

A.3 The Partial JavaMetricsView Class. ................................. 178

A.4 The Expectation of Programming Task B. ......................... 179

A.5 The Partial TextEditorWordCountAction Class. ................. 180

A.6 The Expectation of Programming Task C. ......................... 181

A.7 The Partial AddTraceStatementsEditor Class. ................. 182

A.8 The Expectation of Programming Task D. ......................... 183
LIST OF TABLES

Table                                                Page

3.1  The Quantitative Relaxed Match Values Between the Type
     TextEditor Against its Generalized Types. .................. 29

3.2  The Quantitative Relaxed Match Values between the Type
     WorkbenchPart Against its Specialized Types. ............... 29

3.3  The Quantitative Relaxed Match Values between the Type
     ViewPart Against its Relative Types. ....................... 30

14.1  The Characteristics of 17 Programming Queries. .......... 130

14.2  Distribution of the Best-Fit Ranks for Different Query Types. ...... 141

14.3  Brief Description of the Four Tasks Used for the User Study. ...... 143

14.4  The Characteristics of Participants. ........................... 147

14.5  The Characteristics of 14 Programming Queries Reported in
     Prospector [32]. ................................................... 155
Part I

Introduction
CHAPTER 1
INTRODUCTION

1.1 Motivation

Software reuse, a development process for creating software systems using existing software artifacts rather than building software systems from scratch [27], has been shown to provide quantifiable improvements in both software quality and programmer productivity [4]. This software reuse process typically involves the three key steps [4] shown in Figure 1.1. The first step represents the selection phase of the reuse process during which the best-fit reusable artifacts are selected from possibly heterogeneous repositories based on the system requirements. Once selected, in the second adaptation step, the selected artifacts are adapted to the purpose of the software system. The last step, step 3, corresponds to the integration phase of the reuse process. Here, the adapted artifacts are integrated with on-hand artifacts to create both a complete set of artifacts as well as the final software system.

![Reuse Process Diagram](image)

**Figure 1.1.** The Reuse Process.
Software reuse can be applied at any stage of the software life cycle [16, 27, 3, 15, 24]. Figure 1.2 outlines the reusable software artifacts such as requirement documents, system specifications, design patterns, software units, test cases and development artifacts that are both produced and can potentially be reused at the different stages of the software life cycle. However, while all of these artifacts are reused to some extent, the most commonly reused artifact is a software unit, where a software unit is a clearly delineated piece of software that performs a useful function within a software system [33, 21, 28, 43]. These reusable software units may be of radically different sizes [55, 24] ranging from code fragments, functions, modules, to entire application systems. Code fragments are intrinsic components of the textual source code and are typically available as ascii text, while the other software units are mostly available in the binary format. We refer to these broad categories of software units as code sample (all ascii code units) and components (all binary units).

![Software Life Cycle with Reusable Software Artifacts](image)

**Figure 1.2.** The Software Life Cycle with Reusable Software Artifacts.

In this dissertation, we focus on the procurement of a best-fit software unit that can be re-used within a software system. We use the term “best-fit” to imply a software unit that is determined to be fit for use with respect to the system
requirements and the code under development. Software unit procurement is the first key step of the reuse process particularly at the software implementation stage. The best-fit software unit may range from a library or a framework to individual components in the case of binary software units, to a diverse range of code samples in the case of ascii software units. The procurement of best-fit software units typically follows the two-step process of retrieval and evaluation, highlighted in Figure 1.3. Software unit information based on application requirements is identified as input of the retrieval process. Based on the specified query software unit, the retrieval process searches for candidate software units from its underlying repositories. In the evaluation phase, the software units returned by the retrieval process are ranked to best identify software units with respect to their fitness in the software system.

![Figure 1.3. The Overall Retrieval and Evaluation Processes.](image)

The limited capabilities of current tools available for assisting developers in the retrieval and evaluation process are significant impediments to the successful reusability of large corpus of software units available both within the internal software libraries of corporations and as external widely reputed frameworks and libraries [34, 49, 38, 68, 69, 26, 17, 66, 65, 23, 32, 22]. Assisting developers in procuring the best-fit software units for completing their programming tasks and software systems is thus a significant challenge for the software procurement community.
1.2 Issues in Software Unit Procurement

Procurement of software units is often categorized as component procurement and code sample procurement based on the two broad types of software units. We now detail the issues that currently limit the procurement of component and code artifacts, and summarize the key contributions of previous research in this area.

1.2.1 Component Procurement

Component reuse is the use of existing software components “as is” with no code modification [52, 21, 28, 43]. Particularly, software developers need only know the functionality of a software component and how it interfaces with its environment and need not alter the component’s internal logic and code. Software components are typically bundled with library or framework packages, or are available as in-house and off-the-shelf components.

Previous Research. Many techniques ranging from keyword-based to full-fledged specification-based heuristics have been proposed in the literature [34, 49, 38, 68, 69, 26, 17, 66, 65] to provide effective retrieval of qualified components. The keyword-based approach [34] is simple and flexible, as users simply specify the query as a set of keywords that represent the requirements of the component in which they are interested. This approach while simple is also prone to low accuracy resulting in either too many or too few hits, or in some cases even completely unrelated hits [56]. The faceted approach [49] classifies components based on predefined taxonomies. While this approach provides a better description of components than a pure keyword-based approach, users must be familiar with the classification scheme to effectively retrieve a needed component. Moreover, it is often hard to manage classification schemes when domain knowledge evolves and as a result the component falls into two or more categories [56]. Signature matching approaches [68] decide the match between two given components, the query and library components, based on the
signatures of the methods in the two components. While signature matching uses intrinsic built-in information about the component, that is its type information, it often still returns irrelevant hits [69, 17]. For example, consider the methods `strcpy` and `strcat` in the standard C library. These methods have the same signature but encode different behaviors. The specification matching approach [69, 26], introduced to overcome the problems of signature matching, is based on the method’s pre- and post-conditions that capture the functionality of the method. While specification matching provides more accurate hits, it is often too time-consuming to be practical as its implementation, based on theorem proving techniques, is expensive [17]. Another drawback of the specification approach is the practical lack of pre- and post-conditions in component code. An approach [17] using test cases that capture the partial semantics of the required functionality via method interactions attempts to address these drawbacks. While this approach tends to improve the performance of the discovery process, defining precise test cases that represent the required functionality is often too hard for the users. The comment matching approach [66, 65] mines comments in the code that explain the functionality of a method. The effectiveness of this approach, however, is limited by the difficulty of writing appropriate comments such that similar methods in the repository together with the similarity between the developer’s query and the different components in the repository [23] can be identified.

In the evaluation process, the fitness of the component with respect to the application requirements and the previously developed components co-deployed in the new system must be determined. The International Standard Organization (ISO) [1] describes the general criteria for software product evaluation while others [48, 2] describe techniques that take into account the needs of particular domains. These evaluation approaches typically involve a combination of paper-based studies of the
components, discussion with other users of those components, and hands-on benchmarking and prototyping [7, 20].

![Diagram of Component Repositories]

**Figure 1.4.** The Current Component Discovery and Evaluation.

**Issues.** Today, the burden of searching and evaluating best-fit software components from different repositories typically falls on the individual developer. In particular, a developer must search for software components from different repositories such as componentsource.com, visualmart.com or internal repositories as shown in Figure 1.4, requiring the knowledge of each specific interface and component classification. She must further manually examine and evaluate software components to select the best-fit software component that most closely fits her system requirements. Today, while most widely used search engines such as Google and Yahoo bring together different repositories and provide a unified user interface, their search is not customized for software components [64], resulting in substantial unrelated informa-
tion when developers are searching for software components. There is currently no system that comprehensively addresses the issues of (i) integrating heterogeneous component repositories; (ii) unifying the user interface using which the developers can specify the query component; (iii) efficiently and accurately searching for components; and (iv) effectively evaluating the discovered candidate components.

1.2.2 Code Sample Procurement

Today’s software developer relies on frameworks and libraries, such as C++ libraries, Java packages, Eclipse packages, and in-house libraries [16, 4, 21, 28, 43, 23], that empower them to create high quality, full-featured applications on-time. Due to the complex APIs of reliant frameworks and libraries, it is thus common practice for developers to use code samples to guide their development effort [42, 36]. Code samples are typically available to developers in books, manuals, web sites and existing real-life application code, or as examples bundled with library or framework packages, or is code that the programmer has written previously.

Current search engines bundled with frameworks such as Eclipse [44], search and mine the source code using domain types manually specified by users. However, current search engines rely on text-based mining techniques that limits search and hence results to code fragments that explicitly encapsulate the type being queried. Consider the sequence of method invocations `getPart().getWorkbenchWindow().getSelectionService().addSelectionListener(this)`. Here, an object of the type `ISelectionService` is instantiated during the invocation of the method `getSelectionService()`. However, the type `ISelectionService` itself is not explicitly denoted in this source code, thereby rendering text-based mining insufficient for answering simple queries such as "how to instantiate an object of type `ISelectionService`".

8
Previous Research. There have been some techniques [23, 32, 22] proposed in the literature that provide code sample mining targeted to the context of code being developed by a programmer. Strathcona [23] uses different structural context descriptions (parents, invocations, and types) encapsulated in a class (code under development) to create a set of heuristics. Each heuristic is used to query the code repository, returning a set of code samples where the result context matches the query’s context. The final result is a set of code samples that occur most frequently when applying all proposed heuristics. Strathcona, while a good step forward, does not specialize the heuristics it employs based on the developer’s context and its results straddle the extremities – in some cases providing too many irrelevant results, while in others over-constraining the context to provide too few or no results. Prospector [32] defines a query that describes the desired code in terms of input $T_{in}$ and output types $T_{out}$. A query returns code snippets that instantiate an object of $T_{out}$ from a given input type $T_{in}$. To facilitate this, Prospector mines the Jungloid graph that is created using both API method signatures and a corpus of sample client programs, and consists of chains of objects connected via method calls. The results are ranked by the shortest path from $T_{in}$ to $T_{out}$. While Prospector provides more refined, object-instantiation specific queries than Strathcona, it still returns many irrelevant hits or in some cases too few qualified hits – a direct result of its over-reliance on API information and its restrictions on using only the type context.

Method completion tool [22] defines each method as a 154-dimensional vector: the number of lines of code, the number of arguments, a hash of the return type, the cyclomatic complexity, and the frequency count of each of the 150 Java Language token types. The vector $v_q$ of a method being queried is compared to a set of pre-computed vectors $v_i$ of methods in the example repository, and the best method match is returned to the developer. This approach suffers from some of the same drawbacks as Strathcona – the approach is general to all types of queries and can
not be used to ask a more directed query. Moreover, the approach is inflexible using a complete vector comparison at all times, and potentially missing out on partial matches that may be of use. Like Prospector, this approach limits the context to just the type context and does not make use of other potentially useful context information such as the parent context.

**Issues.** Developers today are often impeded by two key programming hurdles when implementing software systems. They correspond to: (i) **object instantiation** - that is “how to instantiate an object of a specified type”; and (ii) **sequence of method invocations** - that is “what is the right sequence of method invocations that encapsulates the life cycle of the specified type”. Retrieving and evaluating code samples to solve these two programming hurdles often falls on the individual developer. In particular, she needs to search for source code and to also understand the syntactic and semantic information embedded in the source code itself to facilitate the extraction of useful code samples. Once extracted, she needs to evaluate each individual code sample with her code under development to determine its reusability and fitness. Each evaluation attempt may take several hours if she is novice to the framework her system is dependent on. There is currently no assistant system that comprehensively addresses the issues of (i) efficiently and accurately mining for code samples; and (ii) effectively evaluating the discovered candidate code samples with respect to the context of the code under development.

### 1.3 Our Work

In this dissertation, we address the issues underlying the procurement of components and code samples. We focus on the improvement of the **quality** and the **ranking** of the retrieved components and code samples, finding the best-fit components or code samples for the task on hand within the first 10 results. This enables
a developer to quickly evaluate a subset of components and code samples before honing in on the best-fit one for her programming task.

1.3.1 The XCoDE Framework

To address the limitation of current component procurement techniques, we have developed the XCoDE framework that allows developers to query for software components based on the query component interface. Particularly, the XCoDE framework exploits not only the syntactic information in types and signatures, but also the semantic information in labels where such information is embedded as part of the properties, methods and events contained in a component. The XCoDE framework has been developed as a Java application that takes as input a query component interface signifying a set of needed properties, methods and events written in Java; and returns as result a set of candidate components. The developer can view and evaluate candidate components, and subsequently reuse the best-fit component, by either directly using it if it is bundled with reliant frameworks and libraries, downloading it from a web site if it is a freeware off-the-shelf component, or purchasing it if it is a commercial off-the-shelf component. In particular, XCoDE makes the following contributions:

- Description Model: XCoDE presents an XML-based unifying component description model (XCM) [58] that serves as a virtual schema for components conforming to different component models. Specifically, XCM provides a standard for the definition of components that crosscuts the different component models and unifies the variance between different models, aiding both component retrieval and evaluation.

- Matching Algorithm: XCoDE provides an innovative hybrid match algorithm, QoMym [59], to retrieve a set of candidate components with respect to a query component interface. The QoMym algorithm combines different matching
algorithms – type matching algorithm to determine the match between two
given domain types or signatures, and linguistic matching algorithm to define
the match between two given labels. Specifically, these matching algorithms
are combined based on the significant value of each information contained
in a component. For example, for a method, the label typically has more
significance than the signature.

- Match Metric: XCoDE supports a unique Quality of Match (QoM) met-
  ric [57, 59] that measures the “goodness” of a match”. QoM is based on
  a match taxonomy and a weight-based match model that qualitatively and
  quantitatively evaluate a match between a query and a library component.
  These are based on information ranging from the type hierarchy to the labels
  of properties, methods and events of the two components. The combination
  of the match taxonomy and the weight-based match model provides a clear
distinction between two or more matches.

### 1.3.2 The XSnippet Framework

To address the limitation of current code sample mining techniques, we have
developed the XSnippet framework that allows software developers to query for
code samples (or code snippets) based on the context of code under development.
The XSnippet framework is currently targeted for aiding the instantiation of domain
types programming hurdle that is initially often faced by software developers due to
the complex APIs in frameworks and libraries. The instantiation of domain types
ranges from simple constructor invocation, to static method invocation, and to a
complex sequence of methods. The XSnippet framework has been developed as an
Eclipse plugin that extends (i) the Eclipse JDT user interface by adding a new
action “XSnippet: query” to the pop-up menu of the Java editor for initiating the
query. This allows a developer to directly initiate the query made available in the
framework from the code she is working on without writing explicit queries; and (ii) a new view "XSnippet: result" for displaying the returned code snippets. Here, the developer can view and evaluate candidate code snippets, and subsequently reuse the best-fit code snippet by copying and pasting it into the code under development.

In particular, XSnippet makes the following contributions [60]:

- Range of Queries: XSnippet supports a range of object-instantiation specific queries from specialized to generalized. These queries allows developers to switch between a context-independent retrieval of code snippets to various degrees of context-sensitive retrieval.

- Source Code Model: XSnippet presents a graph-based source code model that captures the structure and behavior of a source class and provides a formal model for the code snippet mining. Both structure and behavior are a rich syntactic and semantic resource that enables not only the provision of the code context to facilitate the ranking, but also supports newly defined queries.

- Mining Algorithm: XSnippet provides an innovative code mining algorithm, DFSMINE, that accommodates the range of object-instantiation specific queries by constraining the mining process as needed. Specifically, the DFSMINE algorithm is based on a graph representation of Java source class and mines code snippets that exist either within the scope of an individual method or that are spread across the boundaries of two or more methods.

- Ranking Heuristics: XSnippet supports a set of ranking heuristics from context sensitive to context independent. The context sensitive heuristic ranks a set of code snippets based on the contextual information harnessed from the code being queried, while the context independent heuristics rank code snippets based either on the snippet length or the number of times identical
code snippets written within or across source classes mined from the retrieval process.

1.4 Organization of Dissertation

This dissertation is organized as follows.

- Part II details the XCoDE framework: Chapter 2 describes XCM, our unifying component description model that provides a descriptive superset of information for components conforming to various component models in the market. Chapters 3 defines the match between two values of a given primitive element of a component. Chapter 4 defines a Quality of Match (QoM) metric to measure the “goodness” of a match between two given components. The QoM algorithm is introduced in Chapter 5. The experimental evaluation of XCoDE in given in Chapter 6. We present the state of art in component discovery and evaluation including literature on component matching, schema matching, indexing and ranking in Chapter 7 and summarize this part on XCoDE in Chapter 8.

- Part III provides the XSnippet framework: Chapter 9 outlines the key steps of the XSnippet framework from querying to finally presenting the results to developers. Chapter 10 describes the program context and defines object-instantiation specific queries ranging from specialized to generalized. Chapter 11 provides the formal model that captures the structure and behavior of a Java source code for code snippet mining. Chapter 12 presents the code snippet mining for answering a given user query Q. Chapter 13 defines a set of ranking heuristics from context sensitive to context independent. Experimental evaluation of XSnippet is given in Chapter 14. We present the related
work in the area of code sample mining in Chapter 15 and summarize the part on XSnippet in Chapter 16.

- Part IV gives the conclusions and future work.
Part II

\textit{XCoDE}: An XML-based Component Discovery and Evaluation Framework
In recent years, the *black-box* reuse entailing the use of software components "as is" with no code modification has become increasing popular. It popularity is embedded in the fact that software developers need only know the functionality of a software component and how it interfaces with its environment and need not alter the component’s internal logic and code. This software reuse promises advantages such as reduction in development time, reduction in cost and increase in productivity, spurring the development of both commercial and freeware off-the-shelf components. However, while the wide availability of components is essential for the success of software reuse, retrieving and evaluating the best-fit components for its fitness in the system requirement from a large corpus of available off-the-shelf components has rapidly become a key challenge for software developers. Today, developers are faced with inadequate search tools that are ineffective in aiding the procurement of best-fit components from one or more heterogeneous repositories, based on a given query component or a given set of requirements.

In the second part of this dissertation, we present *XCoDE*, an XML-based component discovery and evaluation framework that allows developers to query a component repository for components based on the system requirements as well as the interoperation of the component with components pre-existing in the software system. In particular, *XCoDE* makes the following contributions: (i) an XML-based unifying component description model (XCM) that serves as the virtual schema of components conforming to different component models; (ii) a disciplined application of different matching algorithms to all aspects of a component thereby exploiting the diversity of semantic and syntactic information inherent in a component; and (iii) a unique *Quality of Match* (QoM) metric that measures the "goodness" of a match based on a *match taxonomy* and a weight-based *match model* that qualitatively and quantitatively classify the match between a query and a library component. We present a set of experiments to validate our approach and evaluate the benefits of
XCoDE. In particular, we measure the match quality using the precision and recall used heavily in the information retrieval domain. Head-to-head comparisons have been made between XCoDE and other state-of-the art algorithms found in literature, namely signature, linguistic and a filtering algorithms.

Roadmap: The rest of this part is organized as follows: Chapter 2 describes XCM, our unifying component description model that provides a descriptive superset of information for components conforming to various component models in the market. Chapters 3 defines the match between two values of a given primitive element of a component. Chapter 4 defines a Quality of Match (QoM) metric to measure the “goodness” of a match between two given components. The QoMym algorithm is introduced in Chapter 5. The experimental evaluation of XCoDE in given in Chapter 6. We present the state of art in component discovery and evaluation including literature on component matching, schema matching, indexing and ranking in Chapter 7 and summarize this part on XCoDE in Chapter 8.
CHAPTER 2
XCM: THE COMPONENT ONTOLOGY CONCEPTS

In the context of component matching, the diversity in component models [41, 53, 54, 47, 5] imposes a restriction that often limits component searching to the features specified for a single component model. To extend component matching to encompass a heterogeneous set of components, we present an XML-based unifying component description model, XCM, that crosscuts the information available in these components that conform to diverse component models [58].

Figure 2.1 represents the XCM hierarchical component structure. Here, a component encapsulates a set of properties, methods and events that define how a component itself interacts with other components. The hierarchical component structure in Figure 2.1 can be represented as an XML document, while the general structure of the description model – the XCM concepts – can be described as an XML Schema given in Figure 2.2.

Property. A property is the named attribute of a component and can be read and/or set by other components to govern a component’s appearance or behavior. A property is described by (i) its syntax comprising of pType - the domain type, access - the access type (readWrite, readOnly and writeOnly) and style - the property style (simple, indexed, bound, and constrained); and (ii) its specification comprising of pName - the property label.

Method. A method, termed as the provided method, encapsulates the behavior of a component that can be triggered by other components. Additionally, a component
may require methods, termed as the *required methods*, from other components to complete its functionality [47, 10]. A method is thus described by (i) its syntax comprising of *status* - the method status (provided or required), and *signature* that is in turn defined by *returnType* - the return domain type and *paraType* - the ordered list of parameter domain types; and (ii) its specification comprising of *mName* - the method label.

**Event.** Finally, an *event* is the message used by a component to communicate with other components and is classified as either *published* or *consumed* [47]. A published event is provided by a component to notify its recipients of an event. The recipients must take appropriate actions. A consumed event is an event subscribed to by the component. The subscribed event exists in other components. An event is described by its syntax comprising of *eType* - the event type, *delivery* - the event delivery (unicast or multicast) and *status* - the event status (published or consumed).

**Example.** Figure 2.3 represents the structure of the component *javax.swing.-JSlider* while Figure 2.4 depicts the XML document conforming to XCM that describes the component *JSlider*. Here, the features of the component *JSlider* include: (i) the properties *maximumValue*, *minimumValue* and *currentValue*; (ii) the
Figure 2.2. The XCM Concept Schema.

methods `getMinimumValue()` and `setMinimumValue()`; and (iii) the events `ChangeEvent`. 

21
Figure 2.3. The Structure of The Component JSlider.

```xml
<?xml version="1.0"?>
<xcm>
  <name>javax.swing.JSlider</name>
  <properties>
    <property access="ReadWrite" style="Simple">
      <pName>maximumValue</pName>
      <pType>int</pType>
    </property>
    <property access="ReadWrite" style="Simple">
      <pName>minimumValue</pName>
      <pType>int</pType>
    </property>
  </properties>
  <methods>
    <method status="Provided">
      <mName>getMinimumValue</mName>
      <desc></desc>
      <returnType>int</returnType>
      <paraType></paraType>
    </method>
    <method status="Provided">
      <mName>setMinimumValue</mName>
      <desc></desc>
      <returnType>void</returnType>
      <paraType>int</paraType>
    </method>
  </methods>
  <events>
    <event delivery="MultiCast" status="publish">
      <eType>javax.swing.event.ChangeEvent</eType>
    </event>
  </events>
</xcm>
```

Figure 2.4. The XCM of the JSlider component.
CHAPTER 3
PRIMITIVE MATCH

The goal of XCoDE is two-fold. First, we want to provide a taxonomy and weight-based match model to classify and rank the match between a query component and a set of library (source) components (described in Chapter 4). Second, we want to provide a disciplined technique to combine different matching techniques to improve the quality of the matches discovered by the system (Section 5). These goals are facilitated by the match between the primitive elements of two given components. In this section, we thus begin by first classifying the primitive match – the match between two corresponding leaf elements (primitive information) captured in the XCM structure, such as the label or the type (Figure 2.1).

The quality of match (QoM) between two given primitive elements is qualitatively categorized as exact (=), relaxed (≈) or no match (≠). We term =, ≈ and ≠ the core match operators and assign a numeric weight to each of these match operators. Quantitatively, the operator = is assigned a weight of 1.0 to indicate an exact match, ≈ a weight between 0.0 and 1.0 to denote a relaxed match, and ≠ a weight of 0.0 to represent a no match. These weights represent the match weight of two given primitive elements, denoted as QoM(εq, εs) where εq and εs are query and source primitive elements, respectively.

Dependent on the type of primitive elements, different matching techniques can be employed to determine the actual match, and hence the classification between the two values of a given primitive element. For example, the label of a property is best matched via a linguistic algorithm. Domain types, on the other hand, are
best compared based on their relationships in the overall type hierarchy, requiring specialized type hierarchy comparison algorithms.

In this section, we provide qualitative and quantitative analysis at the primitive level together with the corresponding match algorithms.

3.1 Label Match

A label, typically representative of natural language, can be classified as either (i) an atomic label - composed of a single word; or (ii) a composite label - composed of multiple words, where the start of each word is distinguished generally by punctuations (for example, purchase-order), case distinction (for example, purchaseOrder), or numeric digits (for example, street1). Typically, no restrictions are applied on the words themselves – they can be a fully-defined dictionary word, an abbreviation, an acronym, or a substring. For example, qty is an abbreviation of quantity; uom an acronym of unitOfMeasure; and addr a substring of address.

3.1.1 Qualitative and Quantitative Analysis

The match between the query label $L_q$ and the source label $L_s$, denoted as $QoM_L(L_q, L_s)$, is computed based on the linguistic hierarchy and is qualitatively denoted as: (i) exact - if they are identical that is exactly the same; (ii) relaxed - if they are either synonyms, hypernyms, have the same stem or are in the form of an abbreviation, an acronym or a substring of the other; or (iii) no match - otherwise.

Quantitatively, $QoM_L(L_q, L_s)$ is a normalized value – between 0.0 and 1.0 and is computed as follows. The two given labels are first parsed into tokens (words). The similarity between the query and source tokens is then measured using a linguistic dictionary and/or other auxiliary information. Finally, $QoM_L(L_q, L_s)$ is computed as the average of the best similarity measure of each query token with a source token and is formally given as:
\[ QoM_L(L_q, L_s) = \frac{LSim_q + LSim_s}{|L_q| + |L_s|} \]  

(3.1)

where

\[ LSim_q = \sum_{l_q \in L_q} [\max_{l_s \in L_s} simMatch(l_q, l_s)] \]  

(3.2)

\[ LSim_s = \sum_{l_s \in L_s} [\max_{l_q \in L_q} simMatch(l_s, l_q)] \]  

(3.3)

where \( l_q \) is a query token of the query label \( L_q \), \( simMatch \) is the similarity measure, such as acronym similarity measure and linguistic similarity measure (lin [30], path, or wup [63]), for a pair of tokens, \(|L_q|\) is the number of query tokens for the query label. The variables \( l_s \), \( L_s \) and \(|L_s|\) are similarly defined for the source tokens. Typically, \( QoM_L(L_q, L_s) = 1.0 \) for an exact match, \( QoM_L(L_q, L_s) = 0.0 \) for a no-match and ranges between 0.0 and 1.0 for a relaxed match.

### 3.1.2 Label Match Algorithm

Based on Equation 3.1, we define the **labelMatch** algorithm that computes the QoM of the two given labels using a linguistic dictionary and other auxiliary information. Figure 3.1 gives the pseudo-code for the algorithm. We first check if there is an identity, acronym, abbreviation, substring, or domain similarity match between the two labels as a whole using a domain-specific, local dictionary that defines the common set of abbreviations, acronyms, and commonly used substring/short hand notations for a given domain. If such a match can’t be determined, we check for each token pair taken from the two labels in a similar manner. For each, token pair, if such a match can’t be determined, we invoke the linguistic similarity algorithm to determine the similarity distance between the two tokens. We use the path linguistic similarity measure – a similarity measure based on the path lengths between concepts, and equal to the inverse of the shortest path length between two concepts.
To determine the path similarity of two words, we use Wordnet::Similarity [46], a freely available tool that measures the semantic similarity and the relatedness between a pair of concepts. The tool provides six measures of similarity including the path measure, and three measures of relatedness, all of which are based on the WordNet [39] lexical database.

3.2 Type Match

3.2.1 Qualitative and Quantitative Analysis

The match between the query type \( t_q \) and the source type \( t_s \), denoted as \( QoM_T(t_q, t_s) \), is computed based on the type hierarchy and is qualitatively denoted as: (i) exact - if they are “identical”, exactly the same; (ii) relaxed - if they have similar features such as fields and methods; or (iii) no match - otherwise. Quantitatively, \( QoM_T(t_q, t_s) \) is a normalized value, between 0.0 and 1.0, with an exact match quantified as 1.0, no match as 0.0 and a relaxed match ranging between 0.0 and 1.0.

A relaxed match between two domain types, \( t_q \) and \( t_s \), is classified as (i) specialized - \( t_q \) is a child or descendant of \( t_s \). Consider the partial class hierarchy in Figure 3.2 as an example. Here, the types EditorPart and AbstractTextEditor are specialization of the type WorkbenchPart; (ii) generalized - \( t_q \) is a parent or ancestor of \( t_s \). As shown in Figure 3.2, the types WorkbenchPart and EditorPart are generalization of the type AbstractTextEditor; or (iii) relative - \( t_q \) is neither specialized nor generalized to \( t_s \) but they share similar features inherited from their most immediate common ancestor. In Figure 3.2, the type ViewPart is relative to the types EditorPart and AbstractTextEditor as they share similar features inherited from their ancestor WorkbenchPart.

As type hierarchy is utilized for this classification of a relaxed match, we believe that the path length inherent in the type hierarchy is significant for the quantitative computation of a relaxed match – the lower the path length between two domain
double labelMatch (String: $L_q$, String: $L_s$)
{
    if isIdentical ($L_q$, $L_s$)
        return 1.0;
    else if isSubstring ($L_q$, $L_s$)
        return 0.9;
    else if isAbbreviation ($L_q$, $L_s$)
        return 0.9;
    else if isAcronym ($L_q$, $L_s$)
        return 0.9;
    else if isSimilarInDomain ($L_q$, $L_s$)
        return 0.7;

    token$_q$ = tokenizer ($L_q$);
    token$_s$ = tokenizer ($L_s$);
    for each $t_q \in$ token$_q$ {
        for each $t_s \in$ token$_s$ {
            if isIdentical ($t_q$, $t_s$)
                match[$t_q$][$t_s$] = 1.0;
            else if isSubstring ($t_q$, $t_s$)
                match[$t_q$][$t_s$] = 0.9;
            else if isAbbreviation ($t_q$, $t_s$)
                match[$t_q$][$t_s$] = 0.9;
            else if isSimilarInDomain ($t_q$, $t_s$)
                match[$t_q$][$t_s$] = 0.7;
            else
                match[$t_q$][$t_s$] = linguisticMatch ($t_q$, $t_s$);
        }
    }
    totalWeight = computeMaxMatch (match);
    QoML = totalWeight / (|token$_q$| + |token$_s$|);
    return QoML;
}

**Figure 3.1.** The Label Match Algorithm. The Method `computeMaxMatch (match)` Corresponds to the Numerator of Equation 3.1.

types, the more the similar features shared by them. Formally, a relaxed match of two types, $QoM_T (t_q, t_s)$, is quantitatively computed as:
Figure 3.2. The Partial Class Type Hierarchy along with Dewey Numbers.

\[ QoM_T(t_q, t_s) = \frac{pathLength(t^*) \times 2}{pathLength(t_q) + pathLength(t_s)} \]  \hspace{1cm} (3.4)

where \( t^* \) denotes (i) \( t_q \), if \( t_q \) is generalized to \( t_s \); (ii) \( t_s \), if \( t_q \) is specialized to \( t_s \); or (iii) \( t_p \) – the most immediate common ancestor of \( t_q \) and \( t_s \), if \( t_q \) is relative to \( t_s \). In addition, \( t_s \) denotes the query type, \( t_s \) the source type, and \( pathLength() \) the path length of a specified type \( t \).

**Example: The Specialized Match.** Consider the path from the type `WorkbenchPart` to the type `AbstractDecoratedTextEditor` in Figure 3.2. Here, the type `TextEditor` is a specialization of all types along this path. Given the match \( QoM_T(t_q, t_s) \) in Equation 3.4, Table 3.1 gives the match values between the type `TextEditor` and all types that it specializes from. Here, the match between the type `TextEditor` and these types gradually increases from its distant ancestor.
WorkbenchPart to its direct parent AbstractDecoratedTextEditor. This is mainly
due to the fact that the ancestor potentially implements less features, thereby cre-
ating a significant difference in the overall features encapsulated in the two types.

<table>
<thead>
<tr>
<th>$t_s$</th>
<th>$\mathcal{M}_T(\text{TextEditor}, t_s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkbenchPart</td>
<td>$(2 * 2)/(7 + 2) = 0.44$</td>
</tr>
<tr>
<td>EditorPart</td>
<td>$(3 * 2)/(7 + 3) = 0.60$</td>
</tr>
<tr>
<td>AbstractTextEditor</td>
<td>$(4 * 2)/(7 + 4) = 0.73$</td>
</tr>
<tr>
<td>StatusTextEditor</td>
<td>$(5 * 2)/(7 + 5) = 0.83$</td>
</tr>
<tr>
<td>AbstractDecoratedTextEditor</td>
<td>$(6 * 2)/(7 + 6) = 0.92$</td>
</tr>
</tbody>
</table>

Table 3.1. The Quantitative Relaxed Match Values Between the Type TextEditor Against its Generalized Types.

Example: The Generalized Match. Consider the path from the type Editor-
Part to the type TextEditor in Figure 3.2. Here, the type WorkbenchPart is
a generalization of all types along this path. Given the match $Q_0 \mathcal{M}_T(t_q, t_s)$ in
Equation 3.4, Table 3.2 gives the match values between the type WorkbenchPart
and all types generalized from it. Here, the match between the type WorkbenchPart
against these generalized types gradually decreases from its direct child Editor-
Part to its distant descendant TextEditor. This is mainly due to the fact that
the descendant potentially implements more additional features, thereby creating a
significant difference in the overall features encapsulated in the two types.

<table>
<thead>
<tr>
<th>$t_s$</th>
<th>$\mathcal{M}_T(\text{WorkbenchPart}, t_s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EditorPart</td>
<td>$(2 * 2)/(2 + 3) = 0.80$</td>
</tr>
<tr>
<td>AbstractTextEditor</td>
<td>$(2 * 2)/(2 + 4) = 0.66$</td>
</tr>
<tr>
<td>StatusTextEditor</td>
<td>$(2 * 2)/(2 + 5) = 0.57$</td>
</tr>
<tr>
<td>AbstractDecoratedTextEditor</td>
<td>$(2 * 2)/(2 + 6) = 0.50$</td>
</tr>
<tr>
<td>TextEditor</td>
<td>$(2 * 2)/(2 + 7) = 0.44$</td>
</tr>
</tbody>
</table>

Table 3.2. The Quantitative Relaxed Match Values between the Type WorkbenchPart Against its Specialized Types.
**Example: The Relative Match.** Consider the path from the type `Workbench-Part` to the type `TextEditor` in Figure 3.2. Here, the type `ViewPart` is relative to all types along this path. Given the match $Q_0 M_T(t_q, t_s)$ in Equation 3.4, Table 3.3 gives the match values between the type `ViewPart` and all relative types in its path. Here, the match between the type `ViewPart` and these relative types gradually decreases from the more generalized type `EditorPart` to more specialized type `TextEditor` along this path. This is mainly due to the fact that the `ViewPart` and the generalized type along such path potentially encapsulates more common features as well as less different features, thereby creating a significant difference in the overall features encapsulated in the two types.

<table>
<thead>
<tr>
<th>$t_s$</th>
<th>$M_T$(ViewPart, $t_s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EditorPart</td>
<td>$(2 \times 2)/(3 + 3) = 0.66$</td>
</tr>
<tr>
<td>AbstractTextEditor</td>
<td>$(2 \times 2)/(3 + 4) = 0.57$</td>
</tr>
<tr>
<td>StatusTextEditor</td>
<td>$(2 \times 2)/(3 + 5) = 0.50$</td>
</tr>
<tr>
<td>AbstractDecoratedTextEditor</td>
<td>$(2 \times 2)/(3 + 6) = 0.44$</td>
</tr>
<tr>
<td>TextEditor</td>
<td>$(2 \times 2)/(3 + 7) = 0.40$</td>
</tr>
</tbody>
</table>

**Table 3.3.** The Quantitative Relaxed Match Values between the Type `ViewPart` Against its Relative Types.

### 3.3 Dewey-based Type

A domain type in Java programming language can be either a *primitive* type or a *reference* type. Primitive types include the special boolean type and all numeric types wherein numeric types are further divided into *integral* types (byte, short, int, long and char) and *floating point* types (float and double). The relationship between these types are captured in the primitive hierarchy. Reference types, on the other hand, include *class* types and *interface* types. A class type is generally defined based on the superclass $C_{sp}$ where a superclass $C_{sp}$ is either the `Object` class or a class that inherits either directly or indirectly from the `Object` class. An interface type, on the other hand, is defined based on zero or more super-interfaces. These two different
types often result in independent class and interface hierarchies, respectively. These hierarchies are, however, connected via an implementation relationship between a class and its implemented interfaces. Here, a class $C$ is thus not only referred by itself and its class ancestors along the class hierarchy, but also the interfaces implemented by the class $C$ and the interface ancestors.

As per Equation 3.4, the path lengths of $t_q$, $t_s$ and the most specific common parent of $t_q$ and $t_s$ must be discovered to determine a match between two given types, $t_q$ and $t_s$. This is accomplished by the path traversal along the type hierarchies. Here, the discovery of the most specific parent takes $\theta(n^2)$ time, while the computation of the path length for two types $t_q$ and $t_s$ has $\theta(mn)$ complexity where $m$ represents the number of type hierarchies and $n$ the number of types in each particular type hierarchy.

Distance matrix is a technique that has been introduced by Doerig [13] to select the most specific method among all possible methods for a particular method invocation reflecting the typing rules imposed by JSL, Java Specification Language [18]. The distance matrix defines quantities bigger than or equal to zero for operations which are allowed between certain types, and defines negative quantities for operations which are not applicable between certain pairs of types. Specifically, for class types, the distance between two identical types is 0; between $\textbf{Object}$ and other class type is $-\infty$; between a class type and its specialized class (subtype) is defined as the number of levels between them where the direct supertype is at distance 1; and between relative classes is $-\infty$. Distance for other domain types are applied in a similar vein. During type checking, the smallest quantity bigger than or equal to zero is the one that needs to be chosen by the type checker. If no such smallest positive quantity is found for a particular method invocation, then a compilation error is reported. To compute the match between a type and supertype, the distance matrix thus computes the match in $\theta(1)$ time. However, to compute the match between

31
two arbitrary types, the distance matrix would require an exhaustive comparison of each supertype of \( t_q \) and \( t_s \) to find the most specific common parent of the two types, resulting in a computation complexity of \( \theta(n^2) \) where \( n \) is the number of types in the distance matrix.

To address the performance drawback of both the distance matrix technique and a naive path traversal algorithm, we propose a dewey numbering scheme [61] that captures the hierarchical relationship between the given types. Specifically, each dewey number encodes the path length as well as all ancestor and relative types. Using the dewey scheme, finding the path length and the most specific common parent can be accomplished in \( \theta(1) \) time. We now provide a brief discussion for the transformation of Java types into dewey numbers.

**Class Types.** Figure 3.2 illustrates the partial class type hierarchy along with the dewey numbers. Here, the `Object` class’s dewey number is assigned an initial dewey number of 1.1 – the first numeric number 1 explicitly indicates the class type while the second one refers the order number of the most generic class type. The dewey numbers for the subclasses of the `Object` class are then computed by concatenating the dewey number of the `Object` class as their direct superclass, the symbol “.” followed by the sibling order number. For example, the dewey numbers of the `Widget` and `WorkbenchPart` classes, the direct subclasses of the `Object` class, are 1.1.1708 and 1.1.1718 respectively. Here, numeric number 1.1 indicates the direct superclass, the `Object` class, while the numeric numbers 1708 and 1718 indicate the local order of the classes as children of the `Object` class. The dewey numbers of other classes in the hierarchy are defined in a similar manner. The number of “.”’s encapsulated in the dewey number indicates the path length of the corresponding domain type in the type hierarchy. Thus, the path length of `WorkbenchPart` is 2 as indicated by the number of “.”’s in its dewey number 1.1.1718.
**Interface Types.** Figure 3.3 illustrates two partial interface type hierarchies of `ISaveablePart` and `ISelection`. The dewey number of each interface is defined in a similar manner to that of a class with the exception that the dewey number of each interface initiates with the numeric 2 to explicitly indicate the interface type. Additionally, an interface may be assigned one or more dewey numbers inherited from its super-interfaces. For example, in Figure 3.3, the `IEditorPart` interface has two direct super-interfaces `ISaveablePart` and `IWorkbenchPart`. It thus has two dewey numbers, 2.331.1 and 2.391.83.1, created based on the dewey numbers of its two super-interfaces. Similarly, the interface `ITextEditor` has two dewey numbers, 2.331.1.4 and 2.391.83.1.4, created to reflect the two dewey numbers of its direct super-interface `IEditorPart`.

![Diagram](image)

**Figure 3.3.** The Partial Interface Type Hierarchies along with the Dewey Numbers.

**Class and Interface Types.** To reflect the interface $I$ implemented by a class $C$, additional dewey number(s) derived from the implemented interface $I$ are assigned to the class $C$. These dewey numbers are defined in a similar manner to
that of a class or interface with the exception that the local order appended to the
derived dewey numbers is a decreasing order initiating from -1, where the local
order here represents the order of the classes directly implementing the interface \( I \).
Consider the partial class hierarchy in Figure 3.2. The class \texttt{AbstractTextEditor}
inherits the class \texttt{EditorPart} and implements two interfaces \texttt{IReusableEditor} and
\texttt{ITextEditor}. The dewey numbers assigned to the class \texttt{AbstractTextEditor}
are not only 1.1.1718.1.1 derived from the dewey number 1.1.1718.1 of class
\texttt{EditorPart}, but also 2.331.1.3.-1 and 2.391.83.1.3.-1 derived from two dewey
numbers 2.331.1.3 and 2.391.83.1.3 of interface \texttt{IReusableEditor}, as well as
2.331.1.4.-1 and 2.391.83.1.4.-1 derived from two dewey numbers 2.331.1.4
and 2.391.83.1.4 of interface \texttt{ITextEditor}. In a similar manner, class \texttt{Status
TextEditor} inheriting class \texttt{AbstractTextEditor} is assigned five dewey numbers
1.1.1718.1.1.1, 2.331.1.3.-1.1, 2.391.83.1.3.-1.1, 2.331.1.4.-1.1 and 2-
.391.83.1.4.-1.1.

**Example.** The class and interface type hierarchies are considered with their dewey
numbers to decide on the match between two types. Consider the class and interface
hierarchies in Figure 3.2 and 3.3. The match between the class \texttt{TextEditor} with
dewey number 2.331.1.4.-1.1.1.1 and interface \texttt{ITextEditor} with dewey number
2.331.1.4 can now be discovered in \( \theta(1) \) time. Here, the class \texttt{TextEditor} is
specialized to the interface \texttt{ITextEditor} as the dewey number of class \texttt{TextEditor}
is an extension of that of interface \texttt{ITextEditor}.

### 3.3.1 Type Match Algorithm

We have developed the \texttt{typeMatch} algorithm that is guided by the qualitative
and quantitative analysis in Section 3.2.1. The \texttt{typeMatch} algorithm determines the
match of two given domain types where each domain type is automatically mapped
to its corresponding dewey number before a match is determined. Figure 3.4 and
3.5 presents the typeMatch algorithm. Recall that a given domain type may have one or more dewey numbers. The maximum match value between two given domain types is returned by the typeMatch algorithm.

```java
float typeMatch (String: tq, String: ts)
{
    ArrayList deweyList_q = deweyInstance.lookupDewey(tq);
    ArrayList deweyList_s = deweyInstance.lookupDewey(ts);

    double max = 0.0;
    for each deweyq ∈ deweyList_q {
        for each dewey_s ∈ deweyList_s {
            double weight = typeMatchHelper(tq, deweyq, ts, dewey_s);
            if (weight > max)
                max = weight;
        }
    }
    return max;
}
```

**Figure 3.4.** The typeMatch Algorithm.
float **typeMatchHelper** (String: $t_q$, String: $dewey_q$, String: $t_s$, String: $dewey_s$) {
    if ($dewey_q == dewey_s$) return 1.0;
    else if (isGeneralized($dewey_q$, $dewey_s$)) {
        $length_q = pathLength(dewey_q)$;
        $length_s = pathLength(dewey_s)$;
        return ($length_q * 2$) / ($length_q + length_s$);
    } else if (isSpecialized($dewey_q$, $dewey_s$)) {
        $length_q = pathLength(dewey_q)$;
        $length_s = pathLength(dewey_s)$;
        return ($length_s * 2$) / ($length_q + length_s$);
    } else if (shareParent($dewey_q$, $dewey_s$)) {
        $t_p = latestParent(dewey_q, dewey_s)$
        $length_p = pathLength(dewey_p)$
        $length_q = pathLength(dewey_q)$
        $length_s = pathLength(dewey_s)$
        return ($length_p * 2$) / ($length_q + length_s$);
    } else return 0.0;
}

**Figure 3.5.** The typeMatchHelper Algorithm.
CHAPTER 4
QOM - QUALITY OF MATCH METRIC

We define a Quality of Match (QoM) metric to measure the “goodness” of a match. In particular, we define a match taxonomy and a weight-based match model that qualitatively and quantitatively classifies the match between a query and a library component based on component information, ranging from its type hierarchy to the labels of properties and methods of the two components, defined in the XCM ontology (Chapter 2).

4.1 The Qualitative Analysis - Match Taxonomy

In this section, we define a match taxonomy that qualitatively provides the quality of match (QoM) between two given components. The match taxonomy classifies the matches at the leaf, internal and root nodes of the XCM hierarchical structure (Figure 2.1) as (i) primitive match - match at the leaf level; (ii) micro match - match at the level of properties (or methods, events); and finally (iv) macro match - match at the level of the component. Each level of these matches is tightly coupled and hence heavily dependent on its lower level. In this section, we provide details of micro match and macro match, while primitive match was described in Chapter 3.
4.1.1 Micro Match

A micro match is the match between two given properties, methods and events \(^1\), and is defined as a match of all its primitive elements. The QoM of a micro match between two given properties is said to be (i) exact (\(=\)) - if all primitive elements of the query property match exactly to those of the source property; (ii) relaxed (\(\approx\)) - if either (a) all primitive elements of the query property have a combination of exact and relaxed matches in the source property, or (b) some (but not all) primitive elements of the query property have matches in the source property; or (iii) no match (\(\neq\)) - otherwise. Figure 4.1 pictorially shows examples of micro matches.

\(\text{Figure 4.1. The Micro Match.}\)

Consider the micro matches shown in Figure 4.2. Here, the method int getMinimum() has (i) an exact match with the method int getMinimum() as their signatures and labels are identical; (ii) a relaxed match with two methods int getMinimumValue() and String getMinimum() as it has a relaxed label and a relaxed

\(^1\)For the rest of the section, we describe the matches based on the methods. The match for the properties and events are defined in a similar manner.
signature match respectively with the two methods; and (ii) no match with the method void setCurrentValue(int) as neither signatures nor their labels match.

![Diagram of micro matches](image)

**Figure 4.2.** The Examples of Micro Matches.

### 4.1.2 Macro Match

A match at the highest level, between two components, is termed a *macro* match. A *macro* match is defined on the basis of the matches of all properties, methods and events. Particularly, the QoM for a macro match is classified based on (1) the number of micro matches; and (2) the quality of the micro matches. Based on the number of micro matches, the QoM of a macro match is classified as either a *total* or a *partial* match. In a total match, all elements (properties, methods and events) of the query component match some or all elements of the source component, while in a partial match some (but not all) elements of the query component match those in the source component. Combining the two criteria, the number of matches and the quality of micro matches, we now define four classifications for the QoM at the macro level: *total exact* (=_T), *total relaxed* (≈_T), *partial exact* (=_P) and *partial relaxed* (≈_P). Figure 4.3 pictorially shows examples of the different levels of macro matches.

Consider the components JSliderFieldPanel and JSlider in Figure 4.4. The components JSliderFieldPanel and JSlider have four and three methods, respectively. The methods `getMinimumValue()`, `getMaximumValue()` and `getCurrentValue()`...
Figure 4.3. The Macro Match.

Value() of the component JSliderFieldPanel match the methods getMinimum(), getMaximum() and getCurrentValue() of the component JSlider in a relaxed, relaxed and exact manner, respectively. The method getFieldWidth() of the component JSliderFieldPanel has no match with any method of the component JSlider. Here, the component JSliderFieldPanel thus has a partial relaxed match with the component JSlider. Conversely, the component JSlider has a total relaxed match with the component JSliderFieldPanel, as all its methods have a match with some methods in JSliderFieldPanel.

4.2 The Quantitative Analysis - Weight-Based Match Model

Based on the qualitative analysis, it is often easy to evaluate when one match is better than the other. For example, an exact match is always better than a relaxed match. However, in some cases such a distinction between the QoM for two or more matches cannot be established as easily. For example, based on qualitative analysis
alone we cannot accurately determine whether a total relaxed match is better than a partial exact match, or one partial exact match is better than another partial exact or even a partial relaxed match. To address this deficiency, in this section, we now present a weight-based match model that quantitatively determines and ranks the QoM. We define this quantitative model at each level of the match taxonomy. In this section, we provide details of the weight-based match model at the micro macro level.

4.2.1 Micro Match Model

Based on the QoM of the primitive matches (Section 3), we now define the quantitative value for the micro match of two methods\(^2\) as the normalized sum of the QoM of all its primitive matches. Formally, the QoM of a micro match, denoted as \(QoM(\alpha_q, \alpha_s)\), is given as:

\[
QoM(\alpha_q, \alpha_s) = \sum V_c QoM(\epsilon_q, \epsilon_s)
\]  

\(\text{Figure 4.4. The Example of Macro Match.}\)

\(^2\)The quantitative value for the micro match of two properties and events is defined in a similar manner.
Here, (i) $\alpha_q$ and $\alpha_s$ are the query and source property; and (ii) $\epsilon_q \in \alpha_q$ and $\epsilon_s \in \alpha_s$ are the query and source primitive elements. Intuitively, it can be observed that not all primitive elements have an equal significance in determining the QoM between a query and a source method. For example, for a method, the label typically has more significance than the method signature. We thus specify $\mathcal{V}_c$ as the significance value of the specified primitive element $\epsilon$. In keeping with this intuition, we assign significance values as absolute numeric numbers to all primitive elements of a property where the total significance value of all primitive elements for a property is 1.0.

### 4.2.2 Macro Match Model

To provide a quantitative value for the macro QoM, we define two measures, micro match weight and cardinality ratio. The micro match weight, denoted as $\mathcal{R}_W(C_q, C_s)$, is the normalized sum of QoM of all micro matches of the component and is given as:

$$
\mathcal{R}_W(C_q, C_s) = \sum_{i=1}^{n} \mathcal{V}_i \sum_{\epsilon} \frac{QoM(\alpha_qi, \alpha_s)}{|C_q|}
$$

(4.2)

Here (i) $C_q$ and $C_s$ are the query and source components; (ii) $i$ is the element type that is the property, method or event type; (iii) $n$ is the number of element types defined in the query component; (iv) $\alpha_qi \in q$ and $\alpha_si \in s$ are the query and source elements for the specified type $i$; (v) $|C_q|$ is the number of query elements for the specified type; and (vi) $\mathcal{V}_i$ is the significance value of the specified element type $i$. For example, the method denoting the behavior of the component would typically have more significance than the property.

The cardinality ratio, denoted as $\mathcal{R}_Q(C_q, C_s)$, is the ratio of the number of micro matches and the cardinality of the query component and is given as:
\[ R_Q(C_q, C_s) = \sum_{i=1}^{n} \nu_i \frac{|(C_{qi})^m|}{|C_{qi}|} \] (4.3)

where \(|(C_{qi})^m|\) is the number of micro matches for the specified element type \(i\).

The QoM of a macro match, denoted as \(QoM(C_q, C_s)\), is now defined as the normalized sum of the micro match weight and its cardinality ratio.

\[ QoM(C_q, C_s) = \frac{R_W(C_q, C_s) + R_Q(C_q, C_s)}{2} \] (4.4)
CHAPTER 5

THE QOMYM ALGORITHM

\[ C \rightarrow C_t \rightarrow \text{QoMym} \rightarrow \{C_n, C_{n-1}, C_{n-2}, \ldots \} \]

\[ C_s \rightarrow \text{macroMatch} \rightarrow \text{microMatch} \rightarrow \text{primitiveMatch} \]

\[ \alpha_s \rightarrow \text{microQoM} \rightarrow \text{macroQoM} \]

\[ \varepsilon_s \rightarrow \text{primitiveQoM} \rightarrow \text{microQoM} \]

**Figure 5.1.** The Overall Execution of QoMym.

The *QoMym* algorithm is a depth-first match algorithm that is guided by the match taxonomy presented in Section 4.1 and is directly based on the match model given in Section 4.2. The overall execution of *QoMym* is depicted in Figure 5.1 and the pseudo-code for the algorithm is given in Figures 5.2 - 5.4. The *QoMym* algorithm first evaluates the match values for all primitive elements, that is all leaf nodes including the label and the type of the component itself. The primitive element matches are evaluated based on their type. For example, a linguistic algorithm is employed as part of the label match algorithm to determine the level of match between two labels. Particularly, our label match algorithm uses a combination of
WordNet [39] and a domain-specific dictionary that includes commonly used abbreviations. Similarly, to match two domain types we have developed a type match algorithm to compare the types along a specified type hierarchy.

Each query primitive element is compared with every source primitive element, and all match values above the threshold are saved. This threshold was determined empirically after running a set of control experiments (see details in Chapter 6). Once all primitive matches are computed, the micro matches are evaluated using the \textit{microMatch} module given in Figure 5.3. All match values above a certain threshold are saved. The micro match values are used to determine a single macro match value, the match value of the query and source components, using the \textit{macroMatch} module given in Figure 5.4. The QoMym module (Figure 5.2) finally returns a set of qualified components that have a macro match value above a given threshold.

The running time for the algorithm lies in $\theta(|R| |C_q| |C_s|)$ where $|R|$ represents the number of components in the repository, and $|C_q|$ as well as $|C_s|$ the cardinality of a query and a source component.

\begin{figure}[h]
\begin{center}
\begin{verbatim}
Set QoMym (Component C_q) {
    result $\leftarrow \emptyset$;
    for each Component C_s $\in$ Repository R {
        macroQoM = macroMatch(C_q, C_s);
        if macroQoM $\ge$ macroThreshold
            result $\leftarrow$ result $\cup$ $\langle C_s, macroQoM \rangle$;
    }
    return result;
}
\end{verbatim}
\end{center}
\caption{The QoMym Algorithm.}
\end{figure}

45
double **microMatch** (Micro α_q, Micro α_s) {
    microQoM ← ∅;

    // ε_q ∈ α_q and ε_s ∈ α_s
    for each corresponding primitive pair (ε_q, ε_s) {
        primitiveQoM ← **primitiveMatch** (ε_q, ε_s);
        sig ← DB.getSignificance (ε_q.getType());
        microQoM ← microQoM + (sig * primitiveQoM);
    }

    if microQoM ≥ microThreshold
        return microQoM;
    else
        return 0;
}

**Figure 5.3.** The **microMatch** Algorithm.

double **macroMatch** (Component C_q, Component C_s) {
    // get source and target micro elements
    query ← C_q.getMicroElements () ;
    source ← C_s.getMicroElements () ;

    initialize microQoM[[C_q]][[C_s]] ;

    // calculate QoM for all (α_q, α_s);
    for each micro α_q ∈ query {
        for each micro α_s ∈ source {
            if α_q.getType() == α_s.getType()
                microQoM[α_q][α_s] ← **microMatch** (α_q, α_s);
        }
    }

    // calculate the submacro QoM
    R_W ← getMicroMatchWeight (microQoM);
    R_Q ← getCardinality (microQoM);
    macroQoM ← (R_W + R_Q) / 2;

    return macroQoM;
}

**Figure 5.4.** The **macroMatch** Algorithm.
CHAPTER 6
EXPERIMENTAL RESULTS

The goal of the QoMym algorithm is to improve the overall match quality of the qualified components retrieved in response to a specified query component. We conducted several experiments to evaluate the potential benefits of the QoMym algorithm over other existing algorithms. In this section, we describe our experimental setup and methodology together with our results.

6.1 Experimental Setup and Methodology

Figure 6.1 illustrates the overall architecture of the QoM system. The QoM system together with QoMym algorithm is implemented in Java (SDK 2.0) and deployed on a standalone PC Pentium IV 2.8 GHz with 512 Mb RAM running Microsoft Windows XP. The QoM system takes a query component as input, matches the query component against a set of library components using the QoMym algorithm, and returns the match results to the user. The repository used for the experiments contained JavaBean components across four domains: 24 GUI components, 12 data collection components, 8 calendar components, and 16 components for testing. These components were automatically introspected, transformed into XCM documents and subsequently loaded into the repository.

Measure Of Match Quality. To evaluate our approach, we compared the manually determined real matches \( R \) for a given match task\(^1\) with the matches \( P \) re-

\(^1\)Here a match task denotes the matching of a query component with the components in the repository to determine the qualified components.
turned by the match algorithm. We determined the true positives, i.e., the correctly identified matches, $I$; the false positives, i.e., the incorrectly identified matches, $F = P \setminus I^2$, and the false negatives, i.e., the missed matches, $M = R \setminus I$. Based on the cardinalities of these sets, the Precision and Recall\(^3\) of the match algorithm was computed.

**Figure 6.1.** The Overall Architecture of the QoM System,

- Precision $= \frac{|I|}{|F|} = \frac{|I|}{|I| + |F|}$ estimates the reliability of the match predictions.

- Recall $= \frac{|I|}{|R|}$ specifies the share of real matches that is discovered by the algorithm.

- Overall $= 1 - \frac{|F| + |M|}{|R|} = \frac{|I| - |F|}{|R|} = \text{Recall} \ast (2 - \frac{1}{\text{Precision}})$ represents a combined measure of match quality, taking into account the post-match effort needed for both removing false matches and adding missed matches.

\(^2\)\(F = P \setminus I\) represents all matches in $P$ excluding the matches in $I$.

\(^3\)Precision and Recall are taken from Information Retrieval literature.
6.2 Determining Threshold and Significance Values

The accuracy of the QoMym algorithm is dependent on the threshold and significance values that are an integral part of the weight-based match model defined in Section 4.2. To determine optimal values for these parameters (threshold and significance), we conducted a set of experiments that randomly compared a set of query components against a small number of library components for different threshold and significance values. The overall match values obtained via the QoMym algorithm for the different threshold and significance values were compared against a manual benchmark that we had setup prior to running the experiments. We then gradually added more library components from different domains to determine if the selected threshold and significance values would hold or would need to be adjusted.

There are four major threshold parameters: label and signature thresholds at the primitive level, as well as micro and macro thresholds at the micro and macro levels respectively. Figure 6.2 - 6.3 depicts the average precision and recall for different label and signature thresholds. Here, the results were obtained by comparing the labels and signatures of a set of query components against those of the components in repository. High precision and recall values are good indicators of the quality of the match. Thus, based on these results, we determined the optimal label threshold to be in the range \(0.7 -- 0.9\), and the optimal signature threshold to be in \(0.6 -- 0.8\).

Next, to determine the optimal values for the significance parameters of label and signature, we fixed the label and signature threshold values, and compared the precision and recall obtained by varying the significance values and micro thresholds. Figure 6.4 shows the precision and recall obtained by the algorithm for different
Figure 6.2. Variance in the precision and recall obtained for different label threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis.

significance values \(^4\) and micro thresholds. We found that for obtaining optimal precision and recall, the significance of the signature should be in the range \(\{0.1 \rightarrow 0.2\}\), while the micro thresholds should be in the range \(\{0.6 \rightarrow 0.8\}\).

Finally, to determine the optimal macro thresholds, we fixed the significance value and the micro threshold, and compared the precision and recall obtained by varying the macro thresholds as shown in Figure 6.5. We found the optimal macro threshold to be in the range \(\{0.8 \rightarrow 0.9\}\).

\(^4\)We show signature significance, but the results can be interpreted for label significance value (label = 1 - signature).
Figure 6.4. Variance in the precision and recall obtained for different signature significance values as well as the micro thresholds. Label and signature thresholds were kept constance at 0.8 and 0.7 respectively.

Figure 6.3. Variance in the precision and recall obtained for different signature threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis.
Figure 6.5. Variance in the precision and recall obtained for different macro values. Signature significance values and micro thresholds were kept constant at 0.2 and 0.7 respectively.

In the subsequent experiments, we fix the label threshold for the QoMym algorithm at 0.8, the signature threshold at 0.7, the micro threshold at 0.7, the macro threshold at 0.8, and the significance values of label and signature at 0.8 and 0.2 respectively.

6.3 QoMym Match Quality

Based on the threshold and significance values determined in Section 6.2, we ran a set of experiments to evaluate the accuracy of the QoMym algorithm. We did this by comparing QoMym with our manual benchmark as well as with other state-of-the-art algorithms found in literature, namely a signature, linguistic, and a filtering algorithm that uses signature matching as a filter prior to applying a linguistic match algorithm. For these experiments, we selected the Calendar component shown in Figure 6.6 as the query component, and compared it to the components
in the repository. Figure 6.7 shows the results in terms of the number of qualified components returned from the library by the different algorithms for the query component Calendar.

```java
Component Calendar {
    void setDay (int)         int getDay ();
    void setMonth (int);      int getMonth ();
    void setYear (int);       int getYear ();
    void setStyle (int);      int getStyle ();
    void setLocale (java.util.Locale); java.util.Local getLocale ();
}
```

**Figure 6.6.** The Query Component - Calendar.

Here the expected number of hits for the Calendar component was 4. The QoMym algorithm performed the best returning exactly 4 components, while the signature-based algorithm performed the worst and returned 43 qualified components. It was interesting to note that the filtering algorithm also did better than linguistic and signature algorithms alone returning 6 qualified components. We found there was no difference in the qualified components returned (in this case) if the order of filtering was varied (that is linguistic followed by signature and vice versa).

Figure 6.8 depicts the precision and recall of the different algorithms. In general, high precision and recall are indicators of high overall quality of the match algorithms. We found that all algorithms had high recall (1.0), that is all algorithms were able to find the expected qualified components. The algorithms however varied in their precision with signature based algorithm having a low precision to the QoMym algorithm having the best precision. The filtering algorithm shows that a combination of the linguistic and signature algorithms can result in higher precision than if the algorithms were used individually. This is in step with the intuitive argument that has been made before [69, 26, 17].

53
**Figure 6.7.** The Number of Qualified Components Returned For the Different Matching Techniques.

**Figure 6.8.** The Match Quality of the Different Matching Techniques.
While filtering is a good first step toward combining different algorithms, the experimental results presented here indicate that a disciplined hybrid combination of the two (and in the future more) algorithms can result in higher overall quality of the retrieved qualified components. This is primarily because the significance values are used to weigh the importance of these algorithms. This allows QoMyM to not only discover matches that may have been missed but to also reject matches that are discovered by the filtering technique as different algorithms work independently.
CHAPTER 7
RELATED WORK

7.1 Match Algorithms

Match algorithms have played a central role in not only software engineering application domains such as component-based development, but also in database application domains such as schema integration, data warehousing, and electronic commerce [50]. Specifically, component match algorithms are employed in the software engineering domain to find the correspondence between two given components, while schema match algorithms are used to determine the correspondence for two given schemata in the database domain. We believe that similarities can be drawn between the component and schema match algorithms.

7.1.1 Component Match Algorithms

Many component match algorithms [34, 49, 38, 68, 69, 26, 17, 56, 66, 65] have been proposed in literature to address the problem of component discovery. These algorithms employ a set of techniques ranging from keyword-based to full-fledged specification-based heuristics. The approaches based on theorem proving techniques[38, 69, 26] provide more accurate results but are inherently expensive, while others[34, 49, 68, 17, 56] trade-off the accuracy of results for better performance.

Matsumoto et al. [34] propose the keyword-based approach wherein a query component is described using a set of keywords. Qualified library components are retrieved if their descriptions contain keywords that are identical to or are synonyms of the query keywords. This approach is simple and flexible as users simply specify
the query as a set of keywords that represents the component requirements in which they are interested. However, there is no specific standard for assigning keywords to describe a component, resulting in a discovery process that has low accuracy and results in too many or too few hits or in some cases even completely unrelated hits.

Prieto-Diaz et al. [49] propose the faceted approach where each component is described via a set of terms that are pre-defined for the classified facets. For example, unix components are described using four facets [49]: the function performed by the component, the objects that are manipulated, the data structure used, and the system to which the function belongs. The qualified library component is retrieved based on either identical terms against the specified query terms, or terms that are conceptually close to the specified query terms based on a pre-defined conceptual distance graph. This approach provides better description of unix components than a pure keyword-based approach. However, users must be familiar with the classification scheme of a particular repository to effectively retrieve a needed component [17]. Moreover, it is often hard to manage classification schemes when domain knowledge evolves and as a result the component falls into two or more categories [56].

Mili et al. [38] introduce a component retrieval technique based on a specification match between the query and library components at the component level. In particular, the specification of a component is represented as a binary relation of input and output pair in first-order predicate logic. Specification matching is done via the specification matcher that employs two theorems: first-order theorem for ordering and theorem-proving technique, Otter, for verification. The qualified library component(s) is retrieved based on either the refinement (the same or subset of) or the intersection of the specifications of the library and query components. In particular, a specification \( Sp \) refines a specification \( Sp' \) if and only if any solution to \( Sp \) is also a solution to \( Sp' \).
Zaremski et al. [68, 69] propose the retrieval of software library (ML) components for a given query component based on both signature and specification matches of methods and hence components. Here, the signature and specification match algorithms are applied in sequence with the signature match algorithm filtering out a set of non-matches before the application of the specification match algorithm. The signature of a method is specified via its return type and an ordered list of parameter types. The library component is considered to be qualified if the signatures of the library and query components are either identical, generalized, specialized, uncurry, or reordered. While signature matching uses intrinsic built-in information about the component, that is its type information, it often still returns irrelevant hits [69, 17]. For example, consider the methods `strcpy` and `strcat` in the standard C library. These methods have the same signature but provide different behaviors. The specification matching approach, introduced to overcome the problems of signature matching, is based on the method’s pre- and post-conditions that capture the functionality of the method in first-order predicate logic. Specification matching is done through the specification matcher that relies on the Larch Prover theorem prover. In specification matching, a component is returned if there is either equivalence or implication between the specifications of the library and query components. A disadvantage of such a sequential combination of the two algorithms is that methods that have different signatures but similar specifications such as provided by overloaded methods may be filtered out.

Jeng et al. [26] also propose specification matching as a method for classification, retrieval, and modification of re-usable components. A software component is specified via pre- and post-conditions in terms of order-sorted predicate logic at the granularity of a method. The relationship between two components is based on the sort (type) information and inference rules used for a logical subsumption test. The qualified library component is retrieved based on either identity, equiva-
lence, or subsumption relationship between the specifications of library and query components.

While the specification-based approaches [38, 69, 26] provide more accurate hits, they are too time-consuming to be practical as their implementations often based on theorem proving techniques is expensive [17]. Moreover, component code typically does not contain fully-defined specifications, making the specification match often impractical. In some cases specifications are defined manually based on the textual component descriptions, resulting in an overall approach that is error-prone, and generally not scalable.

Goguen et al. [17] suggest an approach to address the drawbacks of specification matching. A software component is described using algebraic specifications that include complete syntactic information and partial semantic information. The syntactic information is defined via a set of profiles, keywords and method signatures while the semantic information is specified using a set of equations, termed as test case, stating properties that the functions of a component should satisfy. The search is organized as a ranked multi-level filtering (profile, keyword, signature and equation), where each level yields a ranked set of partial matches. A profile is a sequence of numbers that describes how the sorts (types) associated with an operation are organized. Profile matching is thus based on the total number of occurrences of sorts, the cardinalities of the sort groups, the cardinalities of the unrelated sort groups and on whether the value sort is different from any of the argument sorts. Keyword matching is based on an exact keyword match. Signature matching is based on either identity, generalization or specialization of returned types and ordered lists of parameter types. Semantic matching is based on equation testing. This approach avoids the need for a complex theorem prover and does not require any knowledge of the algebraic specifications from its users. However, while this approach tends to
improve the performance of the discovery process, defining precise test cases that represent the required functionality are often too hard to describe.

Sugumaran et al. [56] propose a semantic-based approach with a natural language interface for component retrieval that utilizes the domain knowledge embedded in domain models and ontologies. Domain models such as the auction domain model contain meta-level knowledge in terms of objectives, processes, actions, actors, and components. Ontology of a specified domain model, on the other hand, translates the objectives, processes and etc into a set of terms, information about the terms as well as relationships between the terms. In particular, the user specifies the initial requirements for the components that he/she is interested in using natural language. Terms (keyword and concepts) from such user requirement are identified using a heuristic-based approach. The domain model is then explored via processes and actions related to objectives that match those terms as well as the synonym terms defined in the ontology. The use of the domain model helps users understand more of the context within which the user is performing the search. Users can thus efficiently specify the desired functional requirement of a query component. This functional requirement is decomposed into specific processes and actions using the domain model. The closeness between the query's action and the component's method descriptions is then determined using the conceptual distance that is calculated based on the number of terms that match or are related. However, syntactic information critical to many aspects of software construction is totally ignored, requiring users to manually explore syntactic details of qualified components that semantically match the query component.

Ye et al. [66, 65] have designed, implemented and evaluated a system CodeBroker, that provides different techniques to address the essential challenges in information delivery: to make the delivered information relevant to the task-at-hand and personalized to the background knowledge of an individual developer. In particular,
CodeBroker integrates active information delivery with a reuse repository system, and runs automatically in the background of program editor (Emacs in their case). CodeBroker automatically queries the repository after a developer writes a comment or a method signature, providing her intentions for a developing method. In response to the query, CodeBroker returns a set of methods (together with their corresponding classes), from the reuse repository, that match the \textit{comment} and/or the \textit{method signature}. The developer can evaluate and select one of the resulting methods that is most appropriate to the ones he aims to develop based on her system requirement. The effectiveness of this approach, however, may be limited by the need to and the corresponding difficulty of writing appropriate comments that comprehensively detail the functionality of a method in terms similar to that of a method in the repository [23].

\subsection{Schema Match Algorithms}

Many schema match algorithms [11, 29, 12] have been proposed in literature to address the problem of schema integration. The core of many of these algorithms employ sophisticated linguistic and structural match techniques [31, 11, 19]. Existing linguistic match algorithms mainly perform name similarity and description similarity, while most of the structural match techniques perform graph matching and constraint based matching. Some enhancements to these algorithms have been proposed, such as machine learning [12, 29] and ontological knowledge base applications [40].

Li et al. [29] have developed a semi-automated semantic integration procedure which utilizes the meta-data specification and data contents at the conceptual schema and data content levels. In particular, a self-organizing classifier algorithm is used to categorize attributes according to (i) their field specifications such as data types, length, and supplemental data (format specification and the existence of con-
straints); and (ii) their data contents such as data patterns, value distributions, and data grouping. A back-propagation learning algorithm is then used to train a neural network to recognize input patterns and determine similarity between attributes. Using neural networks for determining attribute equivalence offers several advantages over methods with fixed rules. For example, neural networks are easier to adapt to new problems and can infer relationships unrecognized by programmers since neural networks are trained, not programmed.

Doan et al. [12] propose the LSD (Learning Source Description) system that uses and extends machine learning techniques to semi-automatically create semantic mappings between the source schemata and the mediated schema. LSD first asks user to provide the semantic mappings for a small set of data sources, then uses these mappings together with the sources to train a set of base learners. Each base learner exploits different information either in the source schemata or in their data. Once the base learners have been trained, LSD finds semantic mappings for a new source schema by applying the base learners and then combining their predictions using a meta-learner. To further improve matching accuracy, LSD incorporates domain constraints as well as user feedback as additional sources of knowledge. They have also developed a novel learner that utilizes the structural information in XML documents.

Madhavan et al. [31] have proposed CUPID that combines linguistic and structural matching techniques to establish correspondences between the schema entities. Matching is carried out in three different phases. The first phase of the CUPID approach is the linguistic match. The linguistic match technique in CUPID uses synonyms, hypernyms, tokenization, tagging and elimination to compare schema elements. It relies on a local domain specific dictionary to obtain the possible matches. In the second phase, CUPID transforms the original input schemata into trees and then performs a bottom up structural match, resulting in structural
similarity between pairs of elements. CUPID’s structural match algorithm employs a tree matching technique where the nodes are categorized into leaf and non-leaf elements. Leaf elements are compared according to their data types. Non-leaf elements inherit the total matching costs of their children and their matching cost is increased or decreased depending upon a preset threshold. CUPID carries out linguistic matching and structural comparison separately with different thresholds set up for each of these algorithms and a final match degree is calculated as a combination of the matching costs of these algorithms applied individually on two input schemata. The third phase of CUPID provides a weighted match based on the linguistic and structural similarity of pairs of elements. This final weighted match is used to determine the actual mapping between two elements. In addition to this basic match strategy, CUPID offers three general optimization strategies, namely leaf-pruning, view and lazy expansion, that can be employed to reduce runtime of the match algorithms.

7.2 Component Repository

While most retrieval approaches [34, 49, 38, 68, 69, 26, 17] focus on discovering the correspondence between two given components, a few approaches [38, 26, 17] specifically consider the issue of repository and other auxiliary structures to facilitate the component discovery process. Mili et al. [38] have proposed a lattice-like repository that represents the generalized and specialized hierarchical relationships of components based on component specifications. The discovery process is in this approach applied in a bottom-up fashion against this repository. This process is extremely time-consuming if the repository structure is wide and shallow. Jeng et al. [26] employ a clustering technique to cluster component hierarchies that represent generalized and specialized relationships between components based on the most general components. They apply the discovery process in a top-down fashion.
This approach is, however, inapplicable for object-oriented and hence component libraries as typically more specialized components are designed and implemented for different purposes. For example, the two components com.toedter.calendar.JCalendar and javax.swing.JButton inherit from the component javax.swing.Component but they are developed for different purposes and domains. Goguen et al. [17] partition components in the repositories based on component profiles – a sequence of integers. Components in each partition, where each partition represents a profile that totally or partially matches a query component’s profile, are retrieved to compare against a query component. This partition represents a basic component index, improving the performance of discovery process. However, the component profile is based solely on the syntactic component information, ignoring components that have similar behavior but different profiles while including components that have the same profile but different behaviors in the discovery process.

In database literature, indexing techniques such as B+ tree and hash trees have been proposed to optimize the performance of query process. Dependent on the database type, different information is employed to perform the indexing. For example, indexes based solely on information content are typically constructed for the relational databases, while indexes for XML databases can be constructed on paths, nodes, structure as well as content of XML documents.

Wang et al [62] propose ViST, a unified index on both structure and content for searching XML documents based on B+ trees. Here, both XML documents and XML queries are represented by structure-encoded sequences, instead of nodes or paths, to form the basic unit of query. The structure-encoded sequence, derived from a prefix traversal of a semi-structured XML document, is a sequence of (symbol, prefix) pairs \( \{(a_1, p_1), (a_2, p_2), ..., (a_n, p_n)\} \), where \( a_i \) represents a node in an XML document tree, and \( p_i \) is the path from the root node to node \( a_i \). (A node includes element, attribute, and data that is converted to integer by hash function). Querying
XML data in this case is thus equivalent to finding subsequence matches, avoiding as many unnecessary join operations as possible in query processing. To speed up the matching process, a suffix-tree-like structure is used to index structure-encoded sequences for non-contiguous matching, and B+ trees are then used to index suffix tree nodes.

Rao et al. [51] propose a way of indexing XML documents and processing twig patterns in an XML database. Every XML document in the database is transformed into a sequence of labels by Prufer’s method that constructs a one-to-one correspondence between trees and sequences. In particular, a Prufer sequence is the sequence of integers and the numbering scheme is used to label an XML document tree. During the query processing, a twig pattern is also transformed into its Prufer sequence and then all occurrences of a twig pattern in the database are discovered by performing subsequence matching on the set of sequences in the database. The set of Prufer sequences in the database are indexed using B+ tree in order to support fast subsequence matching for query processing.

7.3 Evaluation Techniques

An average user typically scans only the first ten to twenty matches returned from the discovery process [14]. To enable users to make the right selection, these matches must be evaluated and ranked based on their fitness with respect to the application requirements and the previously developed components co-deployed in the new system. Many ranking techniques [45, 25, 14, 57] have been proposed in literature to provide a distinction between different matches – the retrieved results. Ranking techniques are generally classified into two categories: (i) match-based ranking techniques; and (ii) non match-based ranking techniques. The match-based ranking techniques [57] typically employ the match algorithms’ match results such as the match taxonomy and match weight value to measure the quality of components.
The non match-based ranking techniques [45, 25, 14], on the other hand, typically use rank criteria (i.e. the link structure and the use relation), that is not part of the match criteria used in match algorithms, to measure the quality of matched components returned from the match algorithms.

Previous work on component retrieval [38, 68, 26, 17] has often applied match-based ranking techniques in concert with the component retrieval techniques. This ranking is accomplished at the coarse-grain level either in a qualitative or a quantitative manner. For example, the approaches [38, 68, 26] provide qualitatively ranked matches in that two components are said to be either exact or relaxed based on their match criteria. Gouguen et al. [17] provide ranked matches in a quantitative manner in that the match between two given components is identified by a value number determined by the match algorithms. Clearly, the ranking of components in either a qualitative or a quantitative manner alone appears to be insufficient as two or more components are likely to fall into the same ranked category or number. In addition, while the ranked matches are often based on the component functionality, other useful information such as the component reliability and maintainability that often influences user decisions is not taken into account.

Tansalarak et al. [57] define a Quality of Match (QoM) metric as the measure of “goodness” of a given match, and provide qualitative and quantitative analysis techniques to evaluate the QoM of two given schemata, independent of the actual match algorithm used. In particular, they propose a first of its kind match taxonomy, a qualitative analysis technique, that categorizes the structural overlap and hence the information capacity of the given schemata. This match taxonomy uses UML as its unifying data model, thereby broadening its applicability to relational, XML and OO schemas which can all be expressed in the UML model. To enable distinction of matches within a given category, they propose a quantitative measure of QoM via a weight-based match model. The match model, based on the structural and
informational aspects of a schema, quantitatively evaluates the quality of match, assigning it an absolute numeric value.

There have been numerous non match-based techniques, primarily stemming from information retrieval, that have been proposed. Page et al. [45] propose PageRank to measure the relative importance or quality of web pages, prioritizing the results of web keyword searches. In particular, the academic citation literature technique has been applied to the web’s hypertextual citation structure, wherein every link (backlink) pointing to a given page is considered as an “academic citation” of that page. The rank of a particular page is given as the sum of the ranks of its backlinks, covering two cases: when a page has many backlinks, and when a page has a few highly ranked backlinks. Here, a page with high rank is said to be more important than the page with low rank.

Drori [14] presents the concept of using mutual references between documents within the list of matched documents as a tool for ranking documents, identifying a relatively small subgroup of documents that best reflect the state-of-the-art in that subject area. Here, mutual references are the documents in the list of search results cited by other documents in the list and they themselves also cite other documents in the list.

Inoue et al [25, 67] propose a novel method of ranking software components, called Component Rank, based on analyzing actual uses relations among the components and propagating the significances of components through the use relations. In particular, a collection of software components is represented as a weighted directed graph whose nodes correspond to the components and edges correspond to the usage relation. The components are Java class files while the use relations are the class inheritance, method invocation, and abstract class implementation relations. Weight of a node is defined as the sum of the weights of all incoming edges while the weight of an edge is the multiplication of the distribution ratio and the
weight of the source node. By this measure, stable classes frequently invoked or inherited by other classes have generally high ranks. However, this approach relies on source code analysis, making the ranking of components error-prone. Furthermore, this approach focuses on the component-level analysis of particular systems rather than the usage across multiple existing systems, biasing the ranking of components away from less invoked components in an application or more specified components that may be used in many applications developed by different developers.
CHAPTER 8
SUMMARY

We present $XCoDE$, a Java application that allows developers to query for components from a component repository based on a given query $Q$. $XCoDE$ is bundled with the QoMym algorithm that uses the match taxonomy as a guide for the algorithm execution, and also utilizes the weight-based match model to calculate the QoM for each element of the two components. The QoMym algorithm offers many advantages over the previously developed component matching approaches. For example, while previous approaches take into account the method signature they often discount the importance of labels\(^1\) that impart valuable semantic information. In our work we exploit not only the semantic information in labels, but also the syntactic semantic information in domain types and signature contained in properties, methods and events that are intrinsic parts of a component. In fact we find that with the combined use of the semantic and syntactic information we are able to achieve higher precision and recall without the performance overhead associated with approaches like specification matching. Moreover, our results suggest that a disciplined combination of different algorithms (linguistic and signature) can provide better overall quality than a naive filter-based approach.

\(^1\)The method label should in general provide semantic information to partially characterize the methods if component developers implement components by following the software development guide.
Part III

XSnippet: An XML-based Snippet Mining and Ranking Framework
Today’s programmer relies on frameworks and libraries, such as C++ libraries, Java packages, Eclipse packages, and in-house libraries [16, 4, 21, 28, 43, 23], that empower the programmer to create high quality, full-featured applications on-time. But this reliance comes at a price – these libraries and frameworks represent a steep learning curve due to the sheer number of methods, classes, and interfaces. For example, J2SE, the Java standard library, contains thousands of classes many of which have extremely complex APIs that prove to be a significant impediment to reusability.

It is common practice for software developers to use examples to guide their development efforts. Examples are used to enable software tasks of various shapes and sizes, such as: How to implement an Eclipse plugin that extends the JDT user interface? How to correctly use java.io.PushBackInputStream? How to instantiate an object of ISelectionService? The importance of examples is often directly proportional to the complexity of the library or framework being used and the developer’s experience with it. This largely unwritten, yet standard practice of “develop by example” is often facilitated by examples bundled with library or framework packages, provided in textbooks, and made available for download on both official and unofficial web sites. The vast number of examples that are embedded in the billions of lines of already developed library and framework code are, however, largely untapped.

In the third part of this dissertation, we present XSnippet, a code assistant system that allows developers to query a sample repository for code snippets relevant to the programming task at hand. In particular, XSnippet makes the following contributions: (i) range of object instantiation-specific queries from specialized to generalized; (ii) source code model that provides a formal model for mining code snippets; (iii) mining algorithm that accommodates the range of queries by constraining the mining process as needed and is based on a graph representation of
Java source code; and (iv) a set of ranking heuristics from context sensitive to context independent. We present a set of experiments to validate our approach and to evaluate the potential benefits of XSnippet. In particular, tradeoffs have been studied, examining coverage and ranking for different types of queries ranging from generalized to specialized queries, and the usability of the system via a user study. Head-to-head comparisons have been made between XSnippet and the Prospector code assistant system [32].

Roadmap: The rest of this part is organized as follows: Chapter 9 outlines the key steps of the XSnippet framework from querying to finally presenting the results to developers. Chapter 10 describes the program context and defines object-instantiation specific queries ranging from specialized to generalized. Chapter 11 provides the formal model that captures the structure and behavior of Java source code for code snippet mining. Chapter 12 presents the code snippet mining for answering a given user query Q. Chapter 13 defines a set of ranking heuristics from context sensitive to context independent. Experimental evaluation of XSnippet is given in Chapter 14. We present the related work in the area of code sample mining in Chapter 15 and summarize the part on XSnippet in Chapter 16.
CHAPTER 9
THE FRAMEWORK OVERVIEW OF XSNIPPET

XSnippet is a code assistant framework that enables developers to retrieve candidate code snippets to solve a particular programming hurdle without writing explicit queries. The XSnippet has been developed as an Eclipse plugin and can be invoked from within the Eclipse Java editor. Figure 9.1 outlines the key steps of the XSnippet framework from querying via the popup menu on the object being queried, to finally presenting the results to the developers via the view “XSnippet: Result” that extends the Eclipse JDT user interface. Here, the developer can view and evaluate candidate code snippets, and reuse the best-fit code snippet by copying and pasting the code snippet into the code under development.

Step 1: Query Formulation. Developers can invoke an XSnippet query from within the Eclipse Java editor environment. Once invoked, the query formulation module extracts the developer request, currently an object instantiation together with the query object type – the domain type $t_q$ of the object that is being queried, and the context – the program context encapsulated in the code under development. This context includes information on the parent (superclass and interfaces) and the lexically visible types. Using this information, the query formulation module constructs the specified “code mining query” that is passed to the snippet mining module.

Step 2: Snippet Mining. The snippet mining module uses the code mining query to first select a set of potential source code model instances from the example
repository $\mathcal{R}$, and then mine each of these source code model instances to produce relevant code snippets. In particular, the snippet mining traverses a set of potential paths encapsulated in the relevant source code model instances using a novel mining algorithm that returns highly qualified code snippets—a sequence of Java programming fragments. Finally, the snippet mining module passes these code snippets to the snippet ranking module.
Step 3: Snippet Ranking. The snippet ranking module ranks the set of code snippets returned by the snippet mining module. The ranking module ranks the code snippets using different ranking heuristics including: (i) the context match – the match between the context of the code snippet and the code under development; (ii) the frequency of code snippets – the number of occurrences of identical code snippets returned by the snippet mining algorithm; and (iii) the length of code snippets – the number of lines of code in the code snippet. A set of ranked code snippets is then returned to the developer in the view “xSnippet: result”. At this point, the developer can select the best-fit code snippet and integrate it in her code to complete her programming task.

In the forthcoming sections, we detail program context and queries in Chapter 10 to support step 1; source code model and snippet mining in Chapter 11 and 12 respectively to support step 2; and snippet ranking to support step 3 in Chapter 13.
CHAPTER 10
PROGRAM CONTEXT AND QUERIES

The context of a program is a rich source of semantic information that can be harnessed to provide better, more targeted code snippets\(^1\) for solving a programming hurdle. Consider the code snippets shown in Figure 10.1. Each of these code snippets instantiate an ICompilationUnit object, albeit with some differences. For example, snippet A instantiates ICompilationUnit from an ISelection object, while snippet E instantiates it from a Map object. The relevance of the code snippet to a developer’s code is dependent on the lexically visible types in the developer’s code. For example, if the developer’s code contains an IEditorPart object, then snippet C is possibly the most relevant snippet as it shows how an ICompilationUnit object can be instantiated from an IEditorPart object.

In this section, we define and classify the context of a developer’s code (or the code context), and then formally define three different types of object-instantiation specific queries based on the code context. We assume that all queries are invoked within method scope.

10.1 Types of Programming Context

A Java source class is defined by its: parents – the superclass it (the source class) inherits from, as well as the interfaces it implements; fields – the named attributes defined within its’ scope; and methods – the class behavior defined at the granularity

\(^1\)We use the term snippets to imply a fragment of code taken from a larger source code file.
A. 
\begin{verbatim}
IStructuredSelection ss = (IStructuredSelection) selection;
Object obj = ss.getFirstElement();
IJavaElement je = (IJavaElement) obj;
IJavaElement ije = je.getAncestor(IJavaElement.COMPILATION_UNIT);
ICompilationUnitCU = (ICompilationUnit) ije;
\end{verbatim}

B. 
\begin{verbatim}
IStructuredSelection ss = (IStructuredSelection) selection;
Object obj = ss.getFirstElement();
IFile f = (IFile) obj;
IJavaElement ie = JavaCore.create(f);
ICompilationUnitCU = (ICompilationUnit) ije;
\end{verbatim}

C. 
\begin{verbatim}
IEditorPart editor;
IEditorInput input = editor.getEditorInput();
IWorkingCopyManager manager = JavaUI.getWorkingCopyManager();
ICompilationUnit CU = manager.getWorkingCopy(input);
\end{verbatim}

D. 
\begin{verbatim}
Object editorInput = SelectionConverter.getInput(editor);
ICompilationUnit unit = (ICompilationUnit) editorInput;
\end{verbatim}

E. 
\begin{verbatim}
Map fMap;
IEditorInput input;
Object obj = fMap.get(input);
ICompilationUnit unit = (ICompilationUnit) obj;
\end{verbatim}

F. 
\begin{verbatim}
JavaPlugin jp = JavaPlugin.getDefault();
IWorkingCopyManager manager = jp.getWorkingCopyManager();
CompilationUnitEditor editor;
IEditorInput iei = editor.getEditorInput();
ICompilationUnit unit = manager.getWorkingCopy(iei);
\end{verbatim}

**Figure 10.1.** Code Snippets that Show Six Different Ways of Instantiating an ICompilationUnit Object.

of methods. We use this basic class structure information to now define *code context* for a given method \( m \) contained in class \( C \).

**Parent Context** \( CT_P (m) \): The parent context of a method \( m \), denoted as \( CT_P (m) \), is a set containing the direct superclass extended by its containing source class \( C \), as well as all interfaces implemented by its containing source class \( C \). Thus, \( CT_P (m) = \{ S, I_i, I_j, \ldots, I_k \} \), where \( m \) is a method \( (m \in C) \) and \( C \) is the source class, \( S \) the direct superclass of \( C \), and \( I_i, I_j \ldots I_k \) the interfaces implemented by \( C \). The parent context \( CT_P \) is a *global* context in that it is shared by all methods \( m_i \in C \).

**Type Context** \( CT_T (m) \): The type context, denoted as \( CT_T (m) \), is the set of types of all visible inherited and local fields, all lexically visible types in the scope of a method \( m \), and the direct superclass extended by its containing source class \( C \).
Thus, \( CT_T (m) = \{ S, \{ t_s \}, \{ t_f \}, \{ t_l \} \} \), where \( m \) is a method \( m \in C \), \( S \) is the direct superclass of \( C \), \( \{ t_s \} \) is the set of types for all visible inherited fields, \( \{ t_f \} \) is the set of types for all local fields, and \( \{ t_l \} \) is the set of all lexically visible types within the scope of the method. The sets \( \{ t_s \} \) and \( \{ t_f \} \) as well as \( S \) define a global context that is shared by all methods \( m_i \in C \), while \( \{ t_l \} \) defines a local context specific to the method \( m \).

Consider that a developer initiates a query from within the method \( m = getCompilationUnit() \) in the JavaMetricsView source class shown in Figure 10.2. Here the superclass ViewPart as well as the two interfaces ISelectionListener and IJavaMetricsListener define the parent context \( CT_P (m) \). The type context \( CT_T (m) \) captures the type Text – the local field’s type in the scope of the source class JavaMetricsView, two types IStructuredSelection and IJavaElement defined and lexically visible in the local scope of the method getCompilationUnit(), the superclass type ViewPart and all visible fields inherited from the ViewPart.

```java
public class JavaMetricsView extends ViewPart
    implements ISelectionListener, IJavaMetricsListener {

    Text message;

    private ICompilationUnit getCompilationUnit( IStructuredSelection ss ) {
        if ( ss.getFirstElement() instanceof IJavaElement ) {
            IJavaElement je = (IJavaElement) ss.getFirstElement();

            ... a developer is currently at this position ..

            return ( ICompilationUnit ) je.getAncestor ( IJavaElement.COMPILATION_UNIT );
        }
    }
}
```

**Figure 10.2.** The JavaMetricsView Class showing the Method getCompilationUnit().

### 10.2 Snippet Queries

Object instantiation is a common problem faced by developers. In the simplest case an object instantiation can be handled by a constructor invocation. Here,
a developer can simply browse the API to comprehend the required parameters. Other more complicated ways of instantiating an object include (i) invocation of a static method, often representing a singleton implementation. For example, the static method invocation JavaUI.getWorkingCopyManager() instantiates an object of type IWorkingCopyManager. In this case, a developer needs to search for methods returning a specified type where a method is defined either in that particular type or in other types that are unknown or unexpected by a developer; and (ii) invocation of a sequence of methods that eventually return the correctly instantiated object. For example, an ISelectionService object can be instantiated by the sequence getVisibleSite().getWorkbenchWindow().getSelectionService() in the class that extends ViewPart. Here, a recursive search for a method returning a specified type is needed. As indicated by these examples, finding the right methods for object instantiation is often not an easy task and causes in many cases developers to become stuck.

To support the procurement of code snippets for the instantiation of a type $t_q$ from within a method $m$, we define a set of queries that range from generalized queries resulting in all possible code snippets that instantiate $t_q$ to specialized context-based queries that result in more specific code snippets for instantiating $t_q$.

**Generalized Instantiation Query $\mathcal{IQ}_G$.** A generalized instantiation query $\mathcal{IQ}_G$ takes as input a type $t_q$ and returns the set of all snippets $s$ contained in the sample code repository that instantiate the type $t_q$. The query $\mathcal{IQ}_G$ is given in Equation 10.1.

$$\mathcal{IQ}_G(t_q) = \forall s \in S : s \text{ contains } t_q \text{ instantiation} \quad (10.1)$$

A generalized instantiation query $\mathcal{IQ}_G$ is useful when there are no code snippets available that meet the requirements of the code context. In such cases, any code
snippet that shows the instantiation of the type \( t_q \) is considered relevant. In general, \( \mathcal{IQ}_G \) returns all code snippets that instantiate \( t_q \). Consider the query \( \mathcal{IQ}_G \) (ICompilationUnit). This query returns all code snippets in the repository, a sampling of which are shown in Figure 10.1.

**A Type-Based Instantiation Query \( \mathcal{IQ}_T \).** A type-based instantiation query \( \mathcal{IQ}_T \) takes as input a type \( t_q \) and the type context \( CT_T (m) \), where \( m \) represents the method in which the query is invoked, and returns the set of all snippets \( s \), contained in the sample code repository, such that the type \( t_q \) is instantiated from some type \( t_b \in CT_T (m) \). Additionally, we extend a type-based query \( \mathcal{IQ}_T \) to return a set of snippets \( s \) where the type \( t_q \) is instantiated from constructor or a static method as their class handles do not typically exist in the type context. The query \( \mathcal{IQ}_T \) \(^2\) can be defined in terms of \( \mathcal{IQ}_G \) as given in Equation 10.2.

\[
\mathcal{IQ}_T(t_q, CT_T(m)) = \exists s \in \mathcal{IQ}_G(t_q) : T(s) \cap CT_T(m) \text{ or } cons/static \rightarrow t_q \quad (10.2)
\]

Here, \( s \) denotes a snippet, \( T(s) \) the lexically visible types in the code snippet \( s \), \( CT_T (m) \) denotes the type context of the method \( m \) as well as \textit{cons} and \textit{static} denote the constructor or static method respectively.

A type-based instantiation query \( \mathcal{IQ}_T \) is useful when (i) the type \( t_q \) can be instantiated independent of the type context, for instance via a constructor or a static method invocation. For example, an \textbf{IWorkingCopyManager} object is often instantiated by the invocation of the static method \textit{getWorkingCopyManager()} of type \textbf{JavaUI} irrespective of whether or not the class \textbf{JavaUI} exists in the context;

\(^2\)To the best of our knowledge, our \( \mathcal{IQ}_T \) is equivalent to the \( \mathcal{IQ}'_T \) defined in the Prospector [32] with the exception that in \( \mathcal{IQ}_T \) the superclass is taken into account.
and (ii) the instantiation of a type \( t_q \) is required from some visible type in the type context. Consider the code segment shown in Figure 10.3. To highlight any selected .java source in the Eclipse package explorer, the method \( \text{selectionChanged}() \) requires the instantiation of an \( \text{ICompilationUnit} \) object from an object of the type \( \text{ISelection} \). Here, \( C_T \) (\( \text{selectionChanged}() \)) = \{\( \text{ISelection, IWorkbenchPart, Text, JavaMetrics, ViewPart} \} \cup \text{all visibly inherited fields from ViewPart}. Based on the code snippets shown in Figure 10.1, the query \( IQ_T \) (\( \text{ICompilationUnit, C_T} \) (\( \text{selectionChanged}() \))) will return code snippets A and B.

```
public class JavaMetricsView extends ViewPart implements ISelectionListener, IJavaMetricsListener {
    Text message;
    JavaMetrics jm;

    public void createPartControl(Composite parent) {
        parent.setLayout(new FillLayout());
        message = new Text(parent, SWT.MULTI);
        message.setText(NO_SELECTION_MESSAGE);
        getSite().getPage().getWorkbenchWindow().getSelectionService().addSelectionListener(this);
        jm = new JavaMetrics();
        jm.addChangeListener(this);
    }

    public void setFocus() {
    }

    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        ICompilationUnit cu;
    }
}
```

**Figure 10.3.** Code Segment of JavaMetricsView Class showing the Object Instantiation of ICompilationUnit.

**Parent-Based Instantiation Query** \( IQ_P \). A well-defined type in a library typically has its own unique functionalities with some dependencies on its parents (direct superclass as well as interfaces). It is our observation that a set of types with the same or similar parents tend to have more relevant code snippets than types that do not share a parent. To reflect the relevance of the shared parents (superclass and/or interfaces), a parent-based instantiation query \( IQ_P \) is defined as follows. A parent-based instantiation query \( IQ_P \) takes as input a type \( t_q \) and the parent
context $CT_P(m)$, where $m \in C$ represents the method in which the query is invoked, and returns the set of all snippets $s$ contained in the sample code repository such that the containing class of the snippet $C_s$ and the class $C$ either inherit from the same class or implement similar interfaces.

$$I_Q_P(t_q, CT_P(m)) = \exists s \in I_Q_G(t_q): CT_P(s) \cap CT_P(m) \quad (10.3)$$

Here, $s$ denotes a snippet, $CT_P(s)$ the parent context of the snippet, $CT_P(m)$ the parent context of the method $m$.

Consider the code segment in Figure 10.4. Here, an ICompilationUnit object in the method $run()$ must capture a .java source file in the Java editor and add some actions to it. Here, $I_Q_T$ (ICompilationUnit, $CT_T$ ($run()$) = {IAction, AddTraceStatementAction} $\cup$ all visibly inherited fields from AddTraceStatementAction) does not provide any code snippets that instantiate ICompilationUnit from IAction and other types. However, ICompilationUnit can be instantiated based on the parent interface IEditorActionDelegate. Thus, while the type-based instantiation query does not provide any results, a parent-based instantiation query $I_Q_P$ (ICompilationUnit, $CT_P$ ($run()$) = {IEditorActionDelegate}) returns the codes snippets $C$ and $D$ shown in Figure 10.1.

```java
public class AddTraceStatementsEditorAction extends AddTraceStatementsAction
    implements IEditorActionDelegate {
    public void run (IAction action) {
        ICompilationUnit cu;
    }

    public void setActiveEditor
        (IAction action, IEditorPart targetEditor) {
    }
}
```

**Figure 10.4.** Code Segment of AddTraceStatementsEditorAction Class showing Object Instantiation of ICompilationUnit.
CHAPTER 11
THE SOURCE CODE MODEL

Source code encapsulates, for most object-oriented languages such as Java and
C++, (i) the class structure: represented by the class inheritance hierarchy, the
implemented interface hierarchy, and the class’s set of members, that is its fields
and methods; and (ii) the class behavior: represented by the computations specified
at the granularity of individual methods. Source code is thus in of itself a rich
syntactic and semantic resource that is not only representative of the system but
that can also be harnessed by other developers to discover viable examples of similar
functionalities in similar contexts.

While text-based source code is the right medium for developers writing code, the
textual format itself has limited applicability for analysis, and for our purpose of min-
ing code snippets. Consider the sequence of method invocations get
ViewPart().get-
WorkbenchWindow().getSelectionService().addSelectionListener(this) in the method
createPartControl() of the JavaMetricsView class shown in Figure 11.13. Here, an
object of the type ISelectionService is instantiated during the invocation of the
method getSelectionService(). However, the type ISelectionService itself is not
explicitly denoted in this source code, thereby rendering text-based mining insuf-
ficient for answering simple queries such as “how to instantiate an object of type
ISelectionService”.

In this section, we define the source code model \(CM\) a directed acyclic graph
\(CM = (N_{CM}, E_{CM})\). The different types of nodes \(N_{CM}\) and edges \(E_{CM}\) defined in
the source code model \(CM\) in concert capture the structure and behavior of a class,
and provide a formal model for the code snippet mining described in later sections. The source code model captures all aspects of the class structure. However, it currently focuses on two primary aspects of the class behavior implementation – *object instantiation* and *method invocation* – to reflect the most common types of queries that we anticipate the developers will ask (see Section 10.2). An instance of the source code model (or simply a CM instance), $cm_s$, corresponds to a source class $C_s$ defined in a single “.java” or “.class” file.

In this section, we describe the different types of nodes $N_{CM}$ and edges $E_{CM}$ of the source code model $CM$, together with a high-level algorithm for transforming a Java source class $C_s$ to a corresponding instance $cm_s$ of the source code model, and the transformation of a mining path embedded in the code model instance to its corresponding code snippet.

### 11.1 Nodes in Source Code Model

A node in the source code model $CM$ can be categorized as *type*, *object* or *method* node. Figure 11.1 gives the graphic representation of these different node types.

![Figure 11.1. The Node Notations defined in the Source Code Model $CM$.](image)

A *type* node can represent (i) a *source class* - the class $C_s$ for the corresponding source code model instance $cm_s$; (ii) a *superclass* - the direct superclass $C_{sp}$ of a specified source class $C_s$; (iii) an *interface* - the interface $i_s \in \mathcal{I}_s$ implemented by the specified source class $C_s$; or (iv) a *generic class* - a class $C_g$ whose static field or method member (including constructor) is referred to or invoked within the specified source...
class $C_s$. A *type* node is labeled by its *domain type*\(^1\) where the domain type is either a class or an interface. Consider the code segment shown in Figure 11.2. The source class `JavaMetricsView`, superclass `ViewPart`, interfaces `ISelectionListener` and `IJavaMetricsListener` are all mapped to *type* nodes and labeled “`JavaMetricsView`”, “`ViewPart`”, “`ISelectionListener`” and “`IJavaMetricsListener`” respectively as shown in Figure 11.2. Additionally in this example, the type `Text`, indicated by the invocation of its constructor, and the type `SWT` (whose static field MULTI is taken as a parameter of the `Text` constructor) are also represented by *type* nodes.

![Diagram](image)

**Figure 11.2.** The Mapping from Java Source Class to *type* Nodes.

An *object* node can represent (i) a *class field* - a field $f_s$ declared within the scope of the source class $C_s$; or (ii) a *method field* - a field $f_m$ declared within the scope of a method $m_s$ where the method $m_s$ is contained in the source class $C_s$. An *object* node is labeled with a *domain type* where the type is a class, an interface or `void`, and a *name* - a label unique within the scope of the method $m_s$ or the source class $C_s$. An exception to this is the name “*this*” used to denote a reference to the source class $C_s$.

\(^1\)We use the full path to represent the domain type. However, in this dissertation, we use the class name for clarification. For example, `ViewPart` represents `org.eclipse.ui.part.ViewPart`. 

85
Consider the sample code in Figure 11.3. Here, the class field `message` is mapped to an `object` node with domain type `Text` and name `message` (Figure 11.3(a)). Similarly, the method fields `parent` and `iww` are transformed into the `object` nodes “Composite parent” (Figure 11.3(b)) and “IWorkbenchWindow iww”, respectively. (Figure 11.3(c)). Note also the invocation of the method `getViewSite()` in the code sample shown in Figure 11.3. The method `getViewSite()` is inherited from the superclass `ViewPart` and hence is implicitly defined in the source class `JavaMetricsView`. This method is invoked using the implicit object `this`. The `this` object is also represented by an `object` node “JavaMetricsView this” created to reflect the object handle for the method invocation (see Figure 11.3(d)). In a similar vein, an object of type `IViewSite` is created to represent the object returned by the method `getViewSite()` and hence the object handle of the method `getWorkbenchWindow()`. As the object is not explicitly denoted in the source class `JavaMetricsView`, we assign a generated name, resulting in the `object` node “IViewSite ivs1”.

Both `type` and `object` nodes can play different roles within a source code model instance `cm`. They can be used to model: `input`, `output`, `parameter` and `return` types and objects. For the rest of this dissertation, to clarify the roles of the `type` and `object` nodes, we use the following notation: (i) input node `n_{t}` refers to a `type` or an `object` node that represents a type or object handle for a method or field value (or address). The `type` node “JavaCore” in Figure 11.4(c) is an example of an input node that represents the class handle for the method `create()”; (ii) output node `n_{o}` refers to an `object` node that is generated by a method invocation, class instance creation, or assignment. The `object` node “IJavaElement ije_1” in Figure 11.4(d) represents

---

2If the object name is not explicitly defined, a default name will be automatically created via the concatenation of all uppercase letters from the domain type together with a numeric number to ensure its uniqueness within the scope of the method `m` or the source class `C_m`. For example, the names of two objects whose domain types are `IViewPart` in a method `m` could be `ivs_1` and `ivs_2`.

86
public class JavaMetricsView extends ViewPart
    implements ISelectionListener, IJavaMetricsListener {

    Text message;

    public void createPartControl(Composite parent) {
        message = new Text  (parent, SWT.MULTI);
        IWorkbenchWindow iww =  getViewSite().getWorkbenchWindow();
    }
}

Figure 11.3. The Mapping from Java Source Class to object Nodes.

an output node generated by the method create(); (iii) parameter node \(n_p\) refers
to a type or an object node used as a parameter of the method \(m_i\) invoked within
the method \(m_s\) or a source code model instance \(cm_s\). The object node “IFile f”
in Figure 11.4(e) represents a parameter node for the method create(); (iv) method
parameter node \(n_{pm}\) refers to a type or an object node used as a parameter explicitly
defined in the signature of a method \(m_s\). The object node “IStructuredSelection
iss” in Figure 11.4(a) represents a method parameter node defined as part of the
signature of the method getIJavaElement(); (v) return node \(n_r\) refers to a type or
an object node returned by the method \(m_s\). The object “IJavaElement tje_1” in
Figure 11.4(d) returned by the invocation of the method JavaCore.create(f)
represents a return node due to the return construct; and (vi) method return node
\(n_{rm}\) refers to a type node that is defined as part of the signature of a method \(m_s\).
The type node “IJavaElement” in Figure 11.4(b) represents a method return node
defined as part of the signature of the method getIJavaElement().

The method return nodes as well as method parameter nodes that are defined
as part of the signature for the method \(m_s\) are annotated with the labels “rm”
and “pm” respectively. Additionally, the label “pm” is combined with a monotonically increasing number to denote the order of the corresponding parameter in the signature. Consider Figure 11.4 once again. The method return type node “IJavaElement” and the method parameter object node “IStructuredSelection iss” are annotated with “rm” and “pm1” respectively to accurately represent the signature of the method getIJavaElement().

![Diagram](image)

**Figure 11.4.** The Roles of Nodes in a Source Code Model Instance $cm_s$.

A *method* node encapsulates the signature as well as the behavior of a method, and is labeled with the method’s name and the method’s modifier (public, private or protected). The method modifiers are represented using the UML notation. The signature of the method is defined by the method’s return type and an ordered list of the method’s parameter types, both of which are represented by *type* and *object* nodes as previously described. The method’s behavior is specified by a set of computations, for example variable declarations and method invocations, that are represented by a source code model sub-instance. Consider the sample code in Figure 11.5. The methods createPartControl() and getCompilationUnit() are mapped to method nodes “+ createPartControl” and “- getCompilationUnit”, respectively.
It should be noted that the method node is the root of a source code model sub-instance that encapsulates both the signature and the behavior of the method.

```java
public class JavaMetricsView extends ViewPart
    implements ISelectionListener, IJavaMetricsListener {
    Text message;
    public void createPartControl(Composite parent) {
        // createPartControl implementation
    }
    private ICompilationUnit getCompilationUnit() {
        // getCompilationUnit implementation
    }
}
```

**Figure 11.5.** The Mapping from Java Source File to Method Nodes.

### 11.2 Edges in Source Code Model

An edge in the source code model $CM$ can be categorized as *inheritance*, *implementation*, *composite*, *method*, *assignment* or *parameter* edge. The *inheritance*, *implementation* and *composite* edges together are used to denote the class structure, while the *method*, *assignment* and *parameter* edges represent the class behavior. Figure 11.6 provides the graphic representation of these different types of edges.

![Edge Notations](image)

**Figure 11.6.** The Edge Notations defined in the Source Code Model.
An inheritance edge represents the relationship between the source class $C_s$ and its direct superclass $C_{sp}$. An implement edge relates the source class $C_s$ to its implemented interface $i_s$. A composite edge denotes the relationship between the source class $C_s$ and its declared class field $f_s \in \mathcal{F}_s$ or method $m_s \in \mathcal{M}_s$ where $\mathcal{F}_s$ represents the set of all fields in $C_s$ and $\mathcal{M}_s$ the set of all methods in $C_s$. Consider the sample code in Figure 11.7(a). Here, the extends construct, specifying inheritance between the source class `JavaMetricsView` and its superclass `ViewPart`, is mapped to an inheritance edge from the type node “`JavaMetricsView`” to type node “`ViewPart`”. The implements construct, specifying the implementation relationship between the source class `JavaMetricsView` and its implemented interfaces `ISelectionListener` and `IJavaMetricsListener`, is mapped to two implement edges from the type node “`JavaMetricsView`” to type node “`ISelectionListener`” and type node “`IJavaMetricsListener`”, respectively. Lastly, the set of class fields and methods in the source class `JavaMetricsView` are mapped to composite edges from the type node “`JavaMetricsView`” to the two object nodes “`Text message`” and “`JavaMetrics jm`”, and to the four method nodes “`+ createPartControl()`”, “`+ setFocus()`”, “`+ selectionChanged()`” and “`- getCompilationUnit()`”. Figure 11.7(b) depicts the class structure for the source code model instance $cm_s$ that is representative of the `JavaMetricsView` source class.

A method edge represents the invocation of a static or non-static method. The edge is outgoing from the type or object input node $n_i$ that represents the type or object handle for the given method $m_i$, and incident on the object node $n_o$ returned by the method $m_i$. The method edge is labeled with the method name to specify the invoked method $m_i$. Consider the sample code in Figure 11.8(a). Here, a static method `getWorkingCopyManager()` declared in the `JavaUI` class is invoked and returns an object of type `IWorkingCopyManager`. This statement is mapped to the method edge “`getWorkingCopyManager()`” that connects the input.
public class JavaMetricsView extends ViewPart implements ISelectionListener, IJavaMetricsListener {
    Text message;
    JavaMetrics jm;
    public void createPartControl(Composite parent) { .. }
    public void setFocus() { .. }
    public void selectionChanged(IWorkbenchPart part, ISelection selection) { .. }
    private ICompilationUnit getCompilationUnit(IStructuredSelection iss) { ¼ }
}

(a) Java source class
(b) class structure

Figure 11.7. The Mapping from Java Source File to Structural Edges.

type node “JavaUI” to output object node “IWorkingCopyManager iwcm_i”. A special case of the method edge is object instantiation via the new construct. The object instantiation in this case is treated as a static method invocation and is represented by a method edge labeled “new” from the type node ni to the object node no. Consider the sample code in Figure 11.8(b). Here, the constructor invocation “new FillLayout()” is mapped to the method edge “new”, connecting the input type node “FillLayout” to the output object node “FillLayout fL1”.

An assignment edge represents the assignment of a value (or address). An assignment statement can be (a) equivalent where the domain types of the output variable vo is the same as the domain type of the input variable vi. In this case, the nodes no and ni corresponding to output and input variable vo and vi respectively are connected via an equivalent edge. Consider the Java statement shown in Figure 11.9 (a). Here, the variables targetEditor and cuEditor are both of the type IEditorPart and are represented as object nodes “IEditorInput targetEditor” and “IEditorPart cuEditor” respectively in the source code model instance.
Figure 11.8. The Mapping from Java Source File to Method Edges.

~cm~. This equivalent assignment is modeled as a directed edge (an equivalent edge) from the input object node “IEditorInput targetEditor” to the output object node “IEditorPart cuEditor”; (b) downcast where the domain type of the output variable v~o~ is more specific than that of the input variable v~i~. This downcasting is represented as a directed edge between the two nodes that represent the two variables, going from the input node n~i~ to the output node n~o~ where n~i~ and n~o~ are the mappings of the input variable v~i~ and output variable v~o~ respectively. Consider the Java statement shown in Figure 11.9 (b). Here, the variable obj is of the type Object. The variable icu_1 of the type ICompilationUnit is created by downcasting type Object to type ICompilationUnit. The variables obj and icu_1 are represented as object nodes in the source code model instance ~cm~, and the downcast is modeled as a downcast assignment edge from the input object node “Object obj” to the output object node “ICompilationUnit icu_1”; and (c) upcast (opposite of a downcast) where the domain type of the output variable v~o~ is more general than that of input variable v~i~. The upcast assignment is modeled by a directed edge between the two nodes n~o~ and n~i~ that model the two variables (v~o~ and v~i~), going from the input node n~i~ (corresponding to the input variable v~i~) to the output node n~o~. Consider
the Java statement in Figure 11.9 (c). Here, the variables \textit{obj} and \textit{icu.1} are of type \texttt{Object} and the more specific type \texttt{ICompilationUnit}, respectively. The variables \textit{obj} and \textit{icu.1} are represented as object nodes in the source code model instance \textit{cm_s} and the upcast assignment is modeled as a directed edge from the input \texttt{object} node “\texttt{ICompilationUnit icu.1}” to output \texttt{object} node “\texttt{Object obj}”.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure11_9.png}
\caption{The Mapping from Java Source File to Assignment Edges.}
\end{figure}

A \textit{parameter} edge represents the participation of a node \textit{n_p} as a parameter of a method \textit{m_i} invoked within the scope of either a method \textit{m_s} or a source code model instance \textit{cm_s}. This edge is unique in that the edge is incident on the method edge that represents the invocation of the method \textit{m_i} requiring the parameter node \textit{n_p}. Consider the code “\texttt{jm.reset(cu)}” in Figure 11.10. The objects \textit{jm}, \textit{cu} and \textit{void} are modeled (as discussed above) as object nodes. Similarly, the method \texttt{reset()} is modeled by a method edge from the input \texttt{object} node “\texttt{JavaMetrics jm}” to output
object node “void”. Here, while the invocation of the method `reset()` is represented by the method edge, we still need to model the input parameters of the method. We do so via a parameter edge. A parameter edge in this case connects the parameter node “ICompilationUnit cu” to the method edge “`reset()`”.

![Diagram](image)

**Figure 11.10.** The Mapping from Java Source File to Parameter Edge.

The method, assignment and parameter edges are annotated with a local order. For the method and assignment edges, the local order is a monotonically increasing number within the scope of a method \( m_s \) and indicates the sequence of computations. For the parameter edges, the local order number is scoped within the method invocation to indicate the order of the parameters in the case of multiple parameters. Assume that the statement “`jm.reset(cu)`” is third in a sequence of statements that capture the behavior of the method. The method edge “`reset()`” is thus annotated with the number 3 as shown in Figure 11.10. In addition, the method `reset()` requires a parameter of type `ICompilationUnit` that is represented by the object node “ICompilationUnit cu”. The parameter edge connecting the object node “ICompilationUnit cu” to the method edge “`reset()`” is also annotated with the number 1 as shown in Figure 11.10.
11.3 Algorithm: Transforming a Java Source Class $C_s$ to the Code Model Instance $CM_s$

Figure 11.11 presents a high-level algorithm, $sourceClass2CM$, for transforming a Java source class, such as the $JavaMetricsView$ class shown in Figure 11.13, to the source code model instance $cm_s$, such as the source code model instance given in Figure 11.12. In the $cm_s$, different types of nodes are used to represent the different types and objects in the source class, and edges represent the relationships between the different nodes.

Specifically, the $sourceClass2CM$ algorithm performs a topdown transformation of the source class to the code model instance initiating with class declaration statement. Type nodes corresponding to the source class $C_s$ being transformed, superclass $C_{sp}$ and interfaces $I_s$ in the class declaration statement are created. Inheritance and implementation edges are created to connect the superclass $C_{sp}$ and interfaces $I_s$ to the source class $C_s$, respectively. This concludes the transformation of the class declaration statement.

The $sourceClass2CM$ algorithm then proceeds sequentially through the source class creating object nodes for each data member (field) encountered. Each object node is connected via a composite edge to the type node corresponding to source class $C_s$ to reflect the containment of the data member in the source class $C_s$.

The $sourceClass2CM$ algorithm next transforms each method $m_s$ contained in the source class $C_s$. A method node for each declared method is created and connected via composite edges to the type node corresponding to source class $C_s$ to reflect the containment of the method in the source class $C_s$. The algorithm then parses the method declaration and creates method return and parameter nodes to reflect its method signature. Each of these nodes (method return and parameter nodes) are encapsulated in the method node. The algorithm proceeds to transform the method behavior encapsulated in the method body. This method behavior is
sourceClass2CM(SourceClass C) {
    create the type node ns representing the source class C;
    if C.getSuperclass() != null {
        create the type node nsp representing the superclass Csp;
        create the inheritance edge, connecting to ns;
    }
    for each is ∈ C.getInterfaces() {
        create the type node nis representing an interface is;
        create the implement edge, connecting ns to nis;
    }
    for each fs ∈ C.getFields() {
        create the object node nfs representing a class field fs;
        create the composite edge, connecting ns to nfs;
    }
    for each ms ∈ C.getMethods() {
        // structure
        create the method node nms representing a method ms;
        create the composite edge, connecting nms to ns;
        // behavior
        create the type node nrtms representing the method return type of ms;
        for each pm ∈ ms {
            create the type/object node npm representing a method parameter pm;
        }
        for each exp ∈ ms {
            expression2CM(exp);
        }
    }
}

**Figure 11.11.** The High-level Algorithm for Transforming a Source Class to Nodes \(N_{CM}\) and Edges \(E_{CM}\) in an Instance of Source Code Model \(cm_s\).

typically represented by a set of statements such as expression, if and return statements. It should be noted that the \(sourceClass2CM\) algorithm currently focuses only on expression statement as an expression is a core statement capable of capturing the two primary aspects of the class behavior – object instantiation and method invocation. Examples of expression statements include a string literal, a number literal, the “this” expression, method invocation, cast expression, class instance creation as well as assignment statements. These expression statements encapsulated in the method behavior are sequentially transformed into nodes and edges. Additionally,
local order annotations are added during the transformation process to capture the sequence of computations.

Figure 11.14 and 11.15 give the high-level algorithms for transforming these expressions. A single object node is created for each literal expression, while two or more nodes together with edges to represent their relationships are created for other expressions such as method invocation, cast expression, class instance creation and assignment (refer to Section 11.1 and 11.2). We now provide a brief discussion of the transformation of these statements.

Method Invocation Statement. For each method invocation, two object nodes \( n_1 \) and \( n_2 \) that represent the method handle and the object returned by the method
public class JavaMetricsView extends ViewPart implements ISelectionListener, IJavaMetricsListener {
    Text message;
    JavaMetrics jm;
    
    public void createPartControl(Composite parent) {
        parent.setLayout(new FillLayout());
        message = new Text(parent, SWT.MULTI);
        message.setText(NO_SELECTION_MESSAGE);
        getViewSite().getWorkbenchWindow().getSelectionService().addSelectionListener(this);
        jm = new JavaMetrics();
        jm.addSelectionListener(this);
    }
    
    public void setFocus() {
    }
    
    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        if (selection instanceof IStructuredSelection) {
            ICompilationUnit cu = getCompilationUnit((IStructuredSelection) selection);
            jm.reset(cu);
        }
    }
    
    private ICompilationUnit getCompilationUnit(IStructuredSelection ss) {
        if (ss.getFirstElement() instanceof IJavaElement) {
            IJavaElement je = (IJavaElement) ss.getFirstElement();
            return (ICompilationUnit) je.getAncestor(IJavaElement.COMPILEATION_UNIT);
        }
        if (ss.getFirstElement() instanceof IFile) {
            IFile f = (IFile) ss.getFirstElement();
            if (f.getFileExtension() != null && f.getFileExtension().compareToIgnoreCase("java") == 0)
                return (ICompilationUnit) JavaCore.create(f);
        }
        return null;
    }
}

Figure 11.13. The Source Class JavaMetricsView.

respectively are created and connected via a method edge \(e_m\). Object nodes \(n_i, \ldots, n_k\) are created to represent the method’s parameters (if any). Each of the object nodes, \(n_i, \ldots, n_k\), are connected to the previous method edge \(e_m\) via parameter edges \(e_{pi}, \ldots, e_{pk}\). Figure 11.16 schematically details the transformation steps for the method invocation \(\text{parent.setLayout(f1.1)}\). Here, \(\text{parent}\) is the handle of the method \(setLayout()\), \(\text{void}\) is the object returned by the method \(setLayout()\), and \(f1.1\) is a parameter of the method \(setLayout()\).

A method invocation expression may encapsulate a sequence of method invocations. Each method invocation is transformed in a similar manner to the method invocation transformation described above. The sequencing of the method invo-
Node expression2CM(Expression exp) {
    if exp is method invocation statement
        Node n = methodInv2CM(exp);
    if exp is class instance creation
        Node n = instance2CM(exp);
    if exp is the cast expression
        Node n = cast2CM(exp);
    if exp is the assignment
        Node n = assignment2CM(exp);
    if exp is the identifier
        create node n for the identifier;
    return n;
}

Node methodInv2CM(MethodInvocation m) {
    for each m_i ∈ m {
        if m_i is the first method invocation
            create type or object node n_i, object node n_o and method edge e_m,
            connecting n_i to n_o;
        else
            create object node n_o and method edge e_m, connecting the previous
            n_o to this n_o;
        for each p ∈ m_i.getParameters() {
            Node n_p = expression2CM(p);
            create parameter edge e_p, connecting n_p to e_m;
        }
    }
    return n_o;
}

Figure 11.14. The High-level Algorithm for Transforming an Expression to Nodes \( \mathcal{N}_{CM} \) and Edges \( \mathcal{E}_{CM} \) in an Instance of Source Code Model \( cm_s \) (1/2).

cations is represented by edges between the object node that corresponds to the object returned by the previous method, and is hence a handle for the next method. Figure 11.17 shows the source code model instance obtained by the transformation of the sequence of method invocations `getViewSite(), getWindowWorkbench(), getSelectionService(), addSelectionListener(this)`.

Class Instance Creation Statement. For a class instance creation statement, a type node representing the constructor handle and an object node representing the object returned by the constructor are created and connected by a method edge \( e_m \). Similar to the method invocation statement, object nodes, \( n_i, ..., n_j \), are created to represent the constructor’s parameters (if any). These parameter nodes, \( n_i, ..., n_k \), are connected to the previous method edge \( e_m \) via corresponding parameter edges.
Node `instance2CM(InstanceCreation i)` {
    create type node `n_i`, object node `n_o` and method edge `e_m`, connecting `n_i` to `n_o`;
    for each `p ∈ i.getParameters()` {
        Node `n_p = expression2CM(p)`;
        create parameter edge `e_p`, connecting `n_p` to `e_m`;
    }
    return `n_o`;
}

Node `cast2CM(CastExpression c)` {
    Node `n_i = expression2CM(c<Expression>())`;
    create object node `n_o` based on `c.getCastType()`;
    create downcast assignment edge `e_ad`, connecting `n_i` to `n_o`;
    return `n_o`;
}

Node `assignment2CM(Assignment assn)` {
    lExp = `assn.getLeftHandSide()`;
    Node `n_o = expression2CM(lExp)`;
    rExp = `assn.getRightHandSide()`;
    Node `n_i = expression2CM(rExp)`;
    if `n_o.getDomainType() == n_i.getDomainType()` {
        create equivalent assignment edge `e_eq`, connecting `n_i` to `n_o`;
    } else {
        create upcast assignment edge `e_uq`, connecting `n_i` to `n_o`;
    }
    return `n_o`;
}

**Figure 11.15.** The High-level Algorithm for Transforming an Expression to Nodes \( N_{CM} \) and Edges \( E_{CM} \) in an Instance of Source Code Model \( cm_s \) (2/2).

---

**Figure 11.16.** The Transformation of the Method Invocation `parent.setLayout(fl_1)`.

Figure 11.18 shows the source code model instance created by the transformation of the class instance creation statement `new FillLayout()`.

**Cast Expression Statement.** Two object nodes, `n_1` and `n_2`, are created for each cast expression. The object nodes `n_1` and `n_2` are connected using a downcast edge `e_d`.

Figure 11.19 shows the source code model instance obtained by the transformation...
Figure 11.17. The Transformation of the Sequence of Method Invocations `getViewSite()`, `getWindowWorkbench()`, `getSelectionService()`, `addSelectionListener(this)`.

Figure 11.18. The Transformation of the Class Instance Creation `new FillLayout()`.

of (IJavaElement) `ss.getFirstElement()`. Here, an object of type `Object` returned by the method `getFirstElement()` is cast to the more specific type `IJavaElement`.

**Assignment Statement.** An assignment expression consists of left and right expressions. The left expression is typically a variable, while the right expression varies and can be for example a method invocation or a cast expression. The left and right expressions are transformed independently of each other, resulting in two final object nodes connected via the assignment edge $e_a$. Figure 11.20 gives the source code model instance obtained by the transformation of the assignment.
Figure 11.19. The Transformation of Cast Expression “(IJavaElement) ss.getFirstElement()”.

The cast statement IJavaElement je = (IJavaElement) ss.getFirstElement(). Here, the object node ije_1 is the result of transforming the right expression (IJavaElement) ss.getFirstElement(). The object node je is obtained by the transformation of the left expression. These two object nodes, ije_1 and je, are connected by the assignment edge e_a from the object node ije_1 to the object node je. It should be noted that for optimized source code model instances and more importantly optimized code snippets, the object node ije_1 is substituted by the object node je to reduce the number of nodes capturing the same information.

11.4 Transforming a Mining Path to the Java Code Snippet

In the previous sections, we have discussed the transformation of a Java source class to the code model instance CM_s. Each potential code model instance CM_s in the sample repository is used in the snippet mining process (details in Chapter 12) to answer a given user query Q. A result returned by the mining process is a code snippet (a sequence of code segments) that is representative of a resulting mining path encapsulated in the code model instance CM_s. A mining path typically consists of one or more sub-path(s) of length 1 where each sub-path encapsulates two nodes connected via a method or an assignment, or a node connected to a method edge via
Figure 11.20. The Transformation of Assignment Statement \[ \texttt{IJavaElement je = (IJavaElement) ss.getFirstElement();} \]

a parameter edge. Each sub-path is transformed to a Java assignment statement and forms the basis of the path transformation to a sequence of assignment statements. In this section, we now provide a brief discussion for the transformation of each sub-path into an assignment statement.

Method Edge. The two nodes, \(n_i\) and \(n_o\), connected via a method edge \(e_m\) are transformed into an assignment statement of the form \(exp = exp_r\) where \(exp\) is a transformation of the node \(n_o\) on which the method edge \(e_m\) is incident on it, and \(exp_r\) is a transformation of the method invocation statement represented by the method annotation on the method edge \(e_m\) and the object or type handle represented by the node \(n_i\). Figure 11.21 shows the transformation of the nodes “JavaUI” and “IWorkingCopyManager iwcm_1” connected via a method edge “getWorkingCopyManager()” to the Java assignment statement IWorkingCopyManager iwcm_1 = JavaUI.getWorkingCopyManager(). It should be noted that if the node \(n_o\) encapsulates the void type, the transformation is simply a method invocation statement. Figure 11.22 shows the transformation of nodes “JavaMetrics jm” and
“void” connected via a method edge “reset()” to the method invocation statement jm.reset(cu).

![Diagram](image)

**Figure 11.21.** The Transformation of Two Nodes Connected via a Method Edge.

![Diagram](image)

**Figure 11.22.** The Transformation of Two Nodes Connected via a Method Edge that is Connected with a Parameter Node via a Parameter Edge.

**Assignment Edge.** Two nodes, $n_i$ and $n_o$, connected via an assignment edge $e_a$ are transformed into an assignment statement $exp_l = exp_r$, where $exp_l$ represents the node $n_o$ on which an assignment edge $e_a$ is incident, and $exp_r$ represents the node $n_i$ with an outgoing assignment edge $e_a$. Recall that an assignment edge can be equivalent, downcast or upcast. The transformation for the different types of assignment edges is as described with the exception that the downcast assignment transformation includes the domain type encapsulated in the $n_o$ and the node $n_i$ in the right expression $exp_r$. Figure 11.23 shows the transformation of two nodes connected via an assignment edge.

104
Parameter Edge. A parameter node $n_p$ and a method edge $e_m$, connected via a parameter edge $e_p$, are transformed into an assignment statement of the form $exp_k = exp_r$. This transformation is two-step process with the first step the transformation of the method edge connecting the two nodes. This is as described above. In the second step, the method is appended the parameter obtained by the transformation of the parameter node $n_p$. Figure 11.22 shows the transformation of a method and parameter edge. It should be noted that if two or more parameter nodes $n_{p1}$, ..., $n_{pk}$ are connected to a method edge $e_m$, these parameter nodes are appended to the method in the order of the local order number annotations on their corresponding parameter edges.
CHAPTER 12
SNIPPET MINING

The goal of the snippet mining, a la the query evaluation, is to mine from a given code sample repository all code snippets that satisfy a given user query $Q$. Figure 12.1 outlines the high level steps undertaken by the snippet mining process. Given the possibly huge number of code samples in the repository, the SelectionAgent pre-selects a set of code model instances $cm_i$ from the code repository based on the user query $Q$. This pre-selection is based in part on the type of the user query ($I_{Q_G}, I_{Q_P}, I_{Q_T}$), and in part on a B+ tree index defined on all types declared or referred to in the code sample repository. For $I_{Q_G}$ and $I_{Q_T}$ queries all code model instances $cm_i$ that have a reference to the query type $t_q$ are pre-selected using the B+ tree index. For $I_{Q_P}$ queries, the pre-selected set of code model instances $cm_i$ is limited to those that refer the query type $t_q$ and those that match the parent context (implement the same interface(s) or extend the same superclass) of the query $Q$.

The pre-selected code instance set $CM$ is passed to the MiningAgent that controls the overall mining process. The MiningAgent invokes the DFSMINE algorithm for every code model instance $cm_i \in CM$. The DFSMINE algorithm, a breadth-first mining algorithm, is the crux of our approach. It (DFSMINE algorithm) traverses a code model instance $cm_i$ and produces as output a set of paths $P$ that represent the final code snippets returned to the user. On completion of the DFSMINE phase, the MiningAgent passes the collection of the sets of paths $\mathcal{P}$ generated by multiple invocations of the DFSMINE algorithm, to the PruningAgent.
Figure 12.1. The snippet mining process.

The PruningAgent removes (i) all duplicate paths $p \in \mathcal{P}$ without modifying their pre-pruning count; and (ii) no-op paths, that is paths that do not encapsulate meaningful functionality. No-op paths include initialization statements such as `ICompilationUnit cu = null`. The pruned mining paths are passed to the SnippetFormulationAgent that transforms each path $p \in \mathcal{P}$ to a corresponding Java code snippet based on the transformation rules outlined in Chapter 11.

The DFSMINE algorithm is a critical component of our approach. We detail this algorithm with examples in the following subsections.

12.1 DFSMINE Algorithm

Figure 12.2 gives the pseudo-code for the DFSMINE algorithm. Given a user query $Q = (t_q, CT)$, where $t_q$ is the query type, and $CT$ the code context in which the query is invoked, and a specific code model instance $cm_i$, the DFSMINE algorithm initiates the mining process by identifying the set of nodes $NQ = \{n_q\}$ such that $n_q$
\( \in cm_i \) and the domain of the node \( n_q \) is \( t_q \) (domain(n_q) = \( t_q \)). The set of nodes \( \mathcal{N}Q \) identifies all instances in the code model instance \( cm_i \) where the query type \( t_q \) is instantiated. In most cases, the node \( n_q \in \mathcal{N}Q \) is scoped within a method instance \( m_i \in cm_i \), and there may be multiple nodes \( n_q \) within the scope of the same method instance \( m_i \). Each such occurrence is treated as a separate instance of the node \( n_q \). Unless stated otherwise, the DFSMINE algorithm is described in the context of one method instance \( m_i \) with the corresponding code model instance \( cm_m \).

```java
private MP DFSMINE (Type \( t_q \), Context CT, CM \( cm_i \)) {
    MP allMPs;
    for each method \( m_i \in cm_i \).getMethods(\( t_q \)) {
        for each node \( n_q \in m_i \).getNodes(\( t_q \)) {
            MP mps = minelQ(\( cm_i \), \( m_i \), \( n_q \), CT);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}
private MP minelQ(CM cm_i, Method \( m_i \), Node \( n_o \), Context CT) {
    if \( n_o \).getDomainType() \( \in \) CT
        return null;
    MP allMPs;
    for each \( n_i \in n_o \).getInputNodes() {
        Edge e = \( n_i \).getAnEdgeTo(\( n_o \));
        MP mps = createAPath(\( n_i \), e, \( n_o \));
        MP nextPaths = minelQ(cm_i, \( m_i \), \( n_i \), CT);
        if nextPaths == null
            allMPs.addPaths(mps);
        else {
            mps.connectPaths(nextPaths);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}
```

**Figure 12.2.** The DFSMINE algorithm.
The goal of the DFSMINE algorithm is to determine for all such instances \( n_q \), types and eventually code segments that instantiate the node \( n_q \) and hence the query type \( t_q \). For each node \( n_q \in NQ \), the DFSMINE algorithm recursively traces back along all edges \( e_i,\ldots,e_j \) incident on the node \( n_q \). This back traversal terminates when it reaches either (i) node \( n_i \in cm_m \), such that there are no edges \( e_i \) incident on it; or (ii) node \( n_i \in cm_m \), such that the domain of the node \( n_i \) is \( t_i \) (domain(\( n_i \)) = \( t_i \)) and \( t_i \) is a type defined in the context \( CT \) of the user query \( Q \) (\( t_i \in CT \)).

Each traversal from \( n_q \) to node \( n_i \) is formulated into a mining path \( p \) such that each path records a forward trace from the termination node \( n_i \) to the initiating node \( n_q \).

The DFSMINE algorithm terminates when all nodes \( n_q \in NQ \) have been traced. The algorithm returns the set of paths \( P \) to the MiningAgent.

**Figure 12.3.** Back Traversal for Query \( IQ = (t_q = ICompilationUnit, CT = \{\}) \). The code model instance of method \( getCompilationUnit() \) in class \( JavaMetricsView \) is shown. The example highlights the basic back traversal of the DFSMINE algorithm.

---

1 Note, this does not include the parameter edge that is incident on a method invocation edge.
To illustrate the working of DFSMINE, consider that the algorithm is invoked with the instantiation query $I Q = (t_q = I C o m p i l a t i o n U n i t, C T = \{ \})$, and the code model instance corresponding to class $JavaMetricsView$ shown in Figure 12.3. In the first phase of the DFSMINE algorithm, we identify the set of nodes $\mathcal{N} Q = \{n_q\}$ such that domain$(n_q) = I C o m p i l a t i o n U n i t$. In this example, $\mathcal{N} Q = \{I C o m p i l a t i o n U n i t \ icu_1\}$, where $n_q = I C o m p i l a t i o n U n i t \ icu_1$ exists in the code model instance of the method $g e t C o m p i l a t i o n U n i t ()$ defined in the $JavaMetricsView$ class instance.

The back traversal phase of the DFSMINE algorithm initiates at the node $n_q = I C o m p i l a t i o n U n i t \ icu_1$. As $I C o m p i l a t i o n U n i t \ icu_1$ has two edges incident on it, the DFSMINE algorithm traces back along both edges to the nodes $I J a v a E l e m e n t \ ije_1$ and $I J a v a E l e m e n t \ ije_2$. At the node $I J a v a E l e m e n t \ ije_1$, the back traversal is applied recursively till it terminates at the node $I S t r u c t u r e d S e l e c t i o n \ ss$. Similarly, the recursive traceback from the node $I J a v a E l e m e n t \ ije_2$ terminates at the node $J a v a C o r e$. This back traversal is shown in Figure 12.3 using broad stroked lines.

As all paths have been traversed, the traversal phase of the algorithm terminates. The DFSMINE algorithm terminates with the completion of the traversal phase, as there are no other nodes $n_q \in \mathcal{N} Q$. On termination the DFSMINE algorithm outputs two paths: $<ss, o_1, je, ije_1, icu_1>$, and $<JavaCore, ije_2, icu_1>$. The corresponding code snippets are shown in Figure 12.4.

In the above example, the back traversal terminates when nodes with no edges incident on them are reached. The traversal can also terminate if a type specified in the context $C T$ of the query is reached. Consider the modified query $I Q = (t_q = I C o m p i l a t i o n U n i t, C T = \{I J a v a E l e m e n t\})$. The query $I Q$ now has a specified type context $C T = \{I J a v a E l e m e n t\}$. The traceback of the paths, in this case, terminates at $I J a v a E l e m e n t \ ije_1$ and $I J a v a E l e m e n t \ ije_2$ respectively resulting

110
1. IStructuredSelection ss;
   Object o_1 = ss.getFirstElement();
   IJavaElement je = (IJavaElement) o_1;
   IJavaElement ije_1 = je.getAncestor();
   ICompilationUnit icu_1 = (ICompilationUnit) ije_1;

2. IFile if_1;
   IJavaElement ije_2 = JavaCore.create(if_1);
   ICompilationUnit icu_1 = (ICompilationUnit) ije_2;

Figure 12.4. Code Snippets for Query $\mathcal{Q} = (t_q = t_{\text{CompilationUnit}}, C_T = \{\})$ based on Code Model Instance in Figure 12.3.

in the final mining paths: $<ije_1,icu_1>$, and $<ije_2,icu_1>$. The corresponding code snippets are shown in Figure 12.5.

1. IJavaElement ije_1;
   ICompilationUnit icu_1 = (ICompilationUnit) ije_1;

2. IJavaElement ije_2;
   ICompilationUnit icu_1 = (ICompilationUnit) ije_2;

Figure 12.5. Code Snippets for Query $\mathcal{Q} = (t_q = t_{\text{CompilationUnit}}, C_T = \{t_{\text{JavaElement}}\})$ based on Code Model Instance in Figure 12.3.

12.2 Extensions to the DFSMINE Algorithm

Consider the example detailed in Figure 12.3. Here for the query $\mathcal{Q} = (t_q = t_{\text{CompilationUnit}}, C_T = \{\})$, the DFSMINE algorithm returns two paths: $p_1 = <ss, o_1, je, ije_1, icu_1>$ and $p_2 = <\text{JavaCore}, ije_2, icu_1>$. While valid, these paths are partial and raise additional questions for the user: How is the object ss obtained? How are parameters of a specific method within the snippet instantiated?
The user, in these cases, would likely need to initiate new queries that search the sample repository for further snippets.

We observe that constraining the traversal phase of the algorithm to (i) edges incident on a node and (ii) within the scope of method boundaries, limits the DFSMINE algorithm to produce only partial snippets in many cases. We now propose extensions to the DFSMINE algorithm to facilitate parameter edge mining as well as across method boundary mining.

**Mining Parameter Edges.** Traversals along a parameter edge $e_{pi}$ incident on a method edge in the traceback path (as defined in the DFSMINE algorithm) are handled similar to all other types of edges. A back traversal along a parameter edge, thus, launches a recursive traceback at its initiating parameter node $n_p$. The key distinction with respect to parameter edges is in the creation of mining paths. While the path itself is constructed as described for the DFSMINE algorithm, each parameter edge traversal represents a sub-path defined in the context of the primary path. The sub-paths constructed by the parameter edge traversals are appended to the primary path.

The traceback phase and the algorithm terminates as described for main DFSMINE algorithm in Section 12.1.

Consider that the BFSMINE-EXT algorithm (see Figure 12.8 - 12.10) is invoked with the instantiation query $\mathcal{I}Q = (t_q=IClass, CT = \{\})$, and the code model instance corresponding to class `JavaMetricsView` shown in Figure 12.6. The BFSMINE-EXT algorithm traces back over all edges, including the parameter edge $e_{pi}$ incident on the method edge `create()`. Traceback along the parameter edge $e_{pi}$ results in the sub-path `<ss, o2, if1>` that is appended to the primary traceback path `<JavaCore, ije2, icu1>`. Thus, on completion the BFSMINE-EXT algorithm produces two paths: $p_1 = <ss, o1, je, ije1, icu1>$, and $p_2 = <JavaCore, ije2, icu1, ss, o2, if1>$. The corresponding code snippets are shown in Figure 12.7.
Figure 12.6. Back Traversal for Query $I_Q = (t_q = \text{ICompilationUnit}, CT = \emptyset)$. The code model instance of method `getCompilationUnit()` in class `JavaMetricsView` is shown. The example highlights the back traversal of all edges.

1. `IStructuredSelection ss;`  
   `Object o_1 = ss.getFirstElement();`  
   `IJavaElement je = (IJavaElement) o_1;`  
   `IJavaElement ije_1 = je.getAncestor();`  
   `ICompilationUnit icu_1 = (ICompilationUnit) ije_1;`

2. `IStructuredSelection ss;`  
   `Object o_2 = ss.getFirstElement();`  
   `IFile if_1 = (IFile) o_2;`  
   `IJavaElement ije_2 = JavaCore.create(if_1);`  
   `ICompilationUnit icu_1 = (ICompilationUnit) ije_2;`

Figure 12.7. Code Snippets for Query $I_Q = (t_q = \text{ICompilationUnit}, CT = \emptyset)$ based on Code Model Instance in Figure 12.6.

**Mining Across Method Boundaries.** Traversals across method boundaries are enabled under two conditions.

**Case 1:** There exists a method parameter node $n_{pm}$ in the code model instance of the method $m_i$ such that $n_{pm}$ represents the parameter object used in the invocation
private MP BFSMINE-EXT (Type t_q, Context CT, CM cm_i) {
    MP allMPs;
    for each method m_i ∈ cm_i.getMethods(t_q) {
        for each node n_q ∈ m_i.getNodes(t_q) {
            MP mps = mineIQ(cm_i, m_i, n_q, CT);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}

**Figure 12.8.** The BFSMINE-EXT algorithm (1/3).

of the method m_i. The method m_i is assumed to be invoked from within a method m_j. In this case, when the parameter method node n_pm of the method m_i is traced, the traversal of the DFSMINE algorithm is extended to traceback the instantiation of the method parameter node at the invocation point of the method m_i in method m_j. The method parameter node n_pm here provides a “hook” point for tracing back to a hitherto un-traced method.

The termination condition for the traversal and the mining path construction is as described for the core DFSMINE algorithm.

**Case 2:** The method edge e_m in the code model instance of the method m_i encapsulates the invocation of a locally defined method m_j, such that method m_j returns an object 0_j of interest during traversal. The DFSMINE algorithm is automatically invoked for all such local methods. To avoid infinite recursion, this automatic invocation of the DFSMINE algorithm is restricted to hitherto un-traced local methods m_j.

Consider the query \(IQ = (t_q = \text{ICompilationUnit}, CT = \{\})\). Figure 12.11 shows the code model instance for the \text{getCompilationUnit()\} and the \text{selection-Changed()\} methods. With the BFSMINE-EXT algorithm, the traversal phase initiates at nodes \text{ICompilationUnit icu_1\} and \text{ICompilationUnit icu_2\} but does not terminate at the \text{IStructuredSelection ss\} – the parameter method node
private MP mineIQ(CM cmi, Method mi, Node no, Context CT) {
    if no.getDomainType() ∈ CT
        return null;
    MP allMPs;
    for each ni ∈ no.getInputNodes() {
        Edge e = ni.getAnEdgeTo(no);
        MP mps;
        if isLocalMethodCall(cmi, ni, e)
            mps = mineCalledMethod(cmi, ni, no, e, CT);
        else
            mps = createAPath(ni, e, no);
        if e.getParameter() ! = null {
            MP paraPaths = mineParameter(cm, m, ni, e, CT);
            mps.connectPaths(paraPaths);
        }
        if ni.isMethodParaNode(mi) {
            MP callerPaths = MineCallerMethod(cm, m, ni, CT);
            mps.connectPaths(callerPaths);
        }
        MP nextPaths = mineIQ(cmi, mi, ni, CT);
        if nextPaths == null
            allMPs.addPaths(mps);
        else {
            mps.connectPaths(nextPaths);
            allMPs.addPaths(mps);
        }
    }
    return allMPs;
}

Figure 12.9. The BFSMINE-EXT algorithm (2/3).

n_{pm} of the method getCompilationUnit(). Instead the algorithm traces back
through the invocation point of the method getCompilationUnit() in the method
selectionChanged() to the instantiation of the getCompilationUnit()’s param-
eter IStructuredSelection iss_1. The back traversal thus terminates at the node
ISelection selection in the-selectionChanged() method. This back traversal is
schematically depicted in Figure 12.11 using broad stroked lines. On completion,
the BFSMINE-EXT algorithm results in two complete paths: \( p_1 = \texttt{<selection, iss_1, ss, o_1, je, ije_1, icu_1>} \), and \( p_2 = \{\texttt{<JavaCore, ije_2, icu_1>}, \texttt{<selection, iss_1, ss, o_2, if_1>}\} \), that are formulated to the code snippets shown in Figure 12.12.

Figure 12.13 illustrates the traceback when the second cross method boundary traversal condition is invoked. Here, we assume that the BFSMINE-EXT algorithm initiates at node \texttt{ICompilationUnit cu} in method \texttt{selectionChanged()}. The method edge incident on the node \texttt{ICompilationUnit cu} encapsulates the invocation of a local method \texttt{getCompilationUnit()}. In this case, the code model instance of the method \texttt{getCompilationUnit()} is automatically mined to provide
more complete code snippets. On termination, the BFSMINE-EXT algorithm produces two mining paths (similar to above): \( p_1 = \langle \text{selection, iss}_1, ss, o_1, je, ije_1, icu_1 \rangle \), and \( p_2 = \langle \text{JavaCore, ije}_2, icu_2 \rangle, \langle \text{selection, iss}_1, ss, o_2, if_1 \rangle \). The corresponding code snippets are similarly shown in Figure 12.12.
1. ISelection selection;
   IStructuredSelection iss_1 = (IStructuredSelection) selection;
   IStructuredSelection ss = iss_1;
   Object o_1 = ss.getFirstElement();
   IJavaElement je = (IJavaElement) o_1;
   IJavaElement ije_1 = je.getAncestor();
   ICompilationUnit icu_1 = (ICompilationUnit) ije_1;

2. ISelection selection;
   IStructuredSelection iss_1 = (IStructuredSelection) selection;
   IStructuredSelection ss = iss_1;
   Object o_2 = ss.getFirstElement();
   IFile if_1 = (IFile) o_2;
   IJavaElement ije_2 = JavaCore.create(if_1);  
   ICompilationUnit icu_1 = (ICompilationUnit) ije_2;

**Figure 12.12.** Code Snippets for Query $I_Q = (t_q = ICompilationUnit, CT = \{\})$ based on Code Model Instance in Figure 12.11.
Figure 12.13. Back Traversal for Query $TQ = (t_q = \text{ICompilationUnit}, CT = \{\})$. The code model instances of methods `getCompilationUnit()` and `selectionChanged()` in class `JavaMetricsView` are shown. The example highlights the back trace across method boundaries via the invocation of a local method.
CHAPTER 13
THE RANKING TECHNIQUE

In general, there is often more than 1 code snippet that can satisfy a given query \( Q \). For example, an object of type \texttt{ICompilationUnit} can be instantiated in many different ways (shown in Figure 13.1), with each corresponding to a potential code snippet returned by an instantiation query \( Q = (t_q = \texttt{ICompilationUnit}) \). The fit of these code snippets in solving the programming task may, however, vary depending on the degree of match between the context of code snippets and the context of the code under development. On average, a user can be expected to scan only the first ten or so code snippets returned by the search process [14]. These code snippets\(^1\) must thus be ranked so as to enable a developer to quickly evaluate a subset of code snippets before honing in on the best-fit code snippet for her programming task. The goal of any ranking heuristic is, therefore, to rank the best-fit code snippets for the task on hand within the first 10 results. In this section, we propose three distinct heuristics to rank the set of code snippets returned by the snippet mining algorithm, namely length, frequency and context-based heuristics.

13.1 Length Ranking

The \textit{length} heuristic ranks the code snippets returned by the snippet mining algorithm based on the number of lines of code encapsulated in the code snippets – the lower the number of lines of code, the higher the rank of the code snippet. This

\(^1\)All code snippets on return are considered to be \textit{true positive} results.
<table>
<thead>
<tr>
<th>Code Snippets</th>
<th>Length H. (a)</th>
<th>Freq. H. (b)</th>
<th>Context H. (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Selection selection;</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>IStructuredSelection ss = (IStructuredSelection) selection;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object obj = ss.getFirstElement();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IJavaElement je = (IJavaElement) obj;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IJavaElement ije = je.getAncestor(1);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit cu = (ICompilationUnit) ije;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B. Selection selection;</td>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>IStructuredSelection ss = (IStructuredSelection) selection;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object obj = ss.getFirstElement();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFile f = (IFile) obj;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IJavaElement ije = javaCore.create(f);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit cu = (ICompilationUnit) ije;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C. EditorPart editor;</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>IEditorInput input = editor.getEditorInput();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IWorkingCopyManager manager = JavaUI.getWorkingCopyManager();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit cu = manager.getWorkingCopy(input);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D. Editor editor;</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Object editorInput = SelectionConverter.getInput(editor);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit unit = (ICompilationUnit) editorInput;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. Map Map</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>IEditorInput input;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object obj = Map.get(input);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit unit = (ICompilationUnit) obj;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F. Plugin.jp = Plugin.getDefault()</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>IWorkingCopyManager manager = jp.getWorkingCopyManager();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CompilationUnitEditor editor;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEditorInput iei = editor.getEditorInput();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICompilationUnit unit = manager.getWorkingCopy(iei);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ICompilationUnit. Figure 13.1. Code Snippets that Instantiate an Object of Type ICompilationUnit.

length heuristic is similar to the shortest path heuristic applied by Prospector [32]. Consider the set of code snippets returned by the instantiation query \( Q = (t_q = \text{ICompilationUnit}, \ C_T = \{\}) \) in Figure 13.1. Figure 13.1(a) shows the ranking of the code snippets using the length heuristic. Here, code snippets \( D \) and \( E \) are ranked above code snippets \( A, B, C \) and \( F \) as they have fewer lines of code.
13.2 Frequency Ranking

A common notion, used in many diverse domains, is that of “frequency” – the more number of times a particular result occurs, the more popular the result and hence deserving of a higher rank. In the context of code snippets, frequency refers to the number of times identical code snippets written within or across different source classes exist in the example repository. Thus, frequency here measures the number of times the identical code snippets are returned by the snippet mining algorithm for the same query \( Q \). Code snippets are ranked based on this frequency measure – the higher the frequency, the higher the rank of the code snippet. For example, if the code snippet \( A \) in Figure 13.1 is returned five times corresponding to five distinct occurrences in the example repository, and code snippet \( F \) has only one occurrence, then code snippet \( A \) is ranked higher than code snippet \( F \), reflecting higher confidence in code snippet \( A \). Figure 13.1(b) gives an example of code snippets ranked based on the frequency heuristic for query \( Q = (t_q = \text{ICompilationUnit}, C_T = \{\}) \).

13.3 Context-based Ranking

The length and frequency ranking heuristics are independent of any context knowledge available in the developer’s code. Intuitively, given the relevance of code snippets to the query \( Q \), it is desirable that these code snippets be ranked high – in the top 10 results. For example, consider the instantiation query \( Q = (t_q = \text{ICompilationUnit}, C_T = \{\text{IEditorPart}\}) \). The code snippet \( C \) in Figure 13.1 is the best-fit result for this query \( Q \) as the instantiation of type \text{ICompilationUnit} is indirectly based on \text{IEditorPart}. However, this code snippet \( C \) is ranked 3 by the length heuristic and 4 by the frequency heuristic. In a similar vein, \text{ICompilationUnit} in code snippets \( D \) and \( F \) are indirectly instantiated from the types \text{JavaEditor} and \text{CompilationUnitEditor} respectively – both specializations of \text{IEditorPart}. 

122
Hence, while not an exact match, the code snippets $D$ and $F$ are relevant to the query $Q$ as the code context (IEditorPart) can be downcasted to either JavaEditor or CompilationUnitEditor when applying these snippets. The length and frequency heuristics, however, rank them as $1$ and $4$, and $3$ and $6$, respectively. A context-sensitive ranking heuristic would rank code snippet $C$, $D$ and $F$ as $1$, $2$, and $3$ respectively as shown in Figure 13.1(c).

We posit that contextual information can be harnessed to provide more effective ranking for a set of code snippets returned by the snippet mining algorithm. We now define the context match measure — a formal measure for the match between the context of a code snippet and the context of a developer’s code, and use this measure to rank the code snippets returned from the snippet mining algorithm.

The context match measure, denoted as $M_{CT}(Q,s)$, is a quantitative measure of how well the parent $CP_s$ and type $CT_s$ contexts of a code snippet $s$ match the parent $CP_q$ and type $CT_q$ contexts specified in the query $Q$. Formally, the context match measure is given as:

$$M_{CT}(Q,s) = \frac{M_P(Q,s) + M_{VT}(Q,s)}{2} \quad (13.1)$$

where $Q$ represents the code mining query formulated by the xSnippet system, $s$ the returned code snippet, $M_P(Q,s)$ the quantitative measure of the match between the direct parents (superclass and interfaces) encapsulated in $Q$ and $s$, and $M_{VT}(Q,s)$ the value quantifying the match between the lexically visible types encapsulated in $Q$ and $s$.

The parent context match $M_P(Q,s)$ is defined as the average of the match between the superclasses $M_S(Q,s)$ and the match between the interfaces $M_I(Q,s)$ of $Q$ and $s$ respectively, and is given as:

$$M_P(Q,s) = \frac{M_S(Q,s) + M_I(Q,s)}{2} \quad (13.2)$$

where
\[ \mathcal{M}_S(Q, s) = \mathcal{M}_T(\text{superclass}(Q), \text{superclass}(s)) \]  
(13.3)

and

\[ \mathcal{M}_I(Q, s) = \frac{I + S}{|\text{inf}(Q)| + |\text{inf}(s)|} \]  
(13.4)

where

\[ I = \sum_{i_q \in \text{inf}(Q)} [\max_{i_s \in \text{inf}(s)} \mathcal{M}_T(i_q, i_s)] \]  
(13.5)

\[ S = \sum_{i_s \in \text{inf}(s)} [\max_{i_q \in \text{inf}(Q)} \mathcal{M}_T(i_s, i_q)] \]  
(13.6)

Here, \text{superclass} and \text{inf} denote the superclass and interfaces respectively encapsulated in either \( Q \) or \( s \); \( i_q \in \text{inf}(Q) \) and \( i_s \in \text{inf}(s) \) are specific instances of the interfaces, \( \mathcal{M}_T(t_c, t_s) \) the match between two given domain types, and \( \mathcal{M}_I(Q, s) \) the average match between the two sets of interfaces – \( \text{inf}(Q) \) and \( \text{inf}(s) \).

The type context match \( \mathcal{M}_{VT}(Q, s) \) defined based on the lexically visible types encapsulated in the query \( Q \) and the types encapsulated in the code snippet \( s \) is given as:

\[ \mathcal{M}_{VT}(Q, s) = \frac{\sum_{t_c \in \text{type}(Q)} [\max_{t_s \in \text{type}(s)} \mathcal{M}_T(t_c, t_s)]}{|\text{type}(Q)|} \]  
(13.7)

where \text{type} denotes the type context encapsulated in \( Q \) and \( s \), \( t_c \in \text{type}(Q) \) and \( t_s \in \text{type}(s) \) are specific types, and \( \mathcal{M}_T(Q, s) \) the match between two given domain types.

The type context match \( \mathcal{M}_{VT} \) is \textit{asymmetric} in that it computes the average of the best similarity of each \( t_c \) with \( t_s \in \text{Type}(s) \) but not necessarily the best similarity of each \( t_s \) with \( t_c \in \text{Type}(Q) \). A symmetric computation of the type context match must consider the match of all new types \( t_s \) that do not exist in or are not relevant to any type in \( \text{type}(Q) \), but that have been relevant in completing a task. The value of the type context match \( \mathcal{M}_{VT} \), in this case, can degrade rapidly if the best similarity
of each $t_s$ with $t_c \in \text{Type}(Q)$ is taken into consideration. For this reason, we have found that an asymmetric match allows for better matches, and hence better ranking of code snippets.

The type match $\mathcal{M}_T(t_c, t_s)$ defined in equations 13.3 - 13.7 is computed using the type match algorithm described in Section 3.2.
CHAPTER 14
EXPERIMENTAL EVALUATION

![Diagram](image)

**Figure 14.1.** The Architectural Overview of the XSnippet framework.

*XSnippet* is a code assistant framework that enables developers to retrieve candidate code snippets for solving a particular programming task (object instantiation) without writing explicit queries. Figure 14.1 gives an architectural overview of the
XSnippet framework, and highlights its three key components, Query Formulation, Snippet Mining and Ranking. The XSnippet system has been developed in Java (SDK 2.0) as an Eclipse plugin, and can be invoked from within the Eclipse Java editor. Figure 14.2 - 14.3 show snapshots of the XSnippet system that extends the Eclipse JDT user interface by adding a new action “XSnippet: query” to the pop-up menu of the Java editor for initiating the query, and a new view “XSnippet: result” for displaying the returned code snippets.

![Figure 14.2. A Snapshot of the XSnippet Interface - Query.](image)

A series of experiments were conducted to evaluate the potential benefits of the XSnippet system. In particular, the experiments were designed to test the following hypotheses:

**Hypothesis 1:** Generalized query, \( \mathcal{I}Q_G \), provides better coverage of tasks than the specialized queries \( \mathcal{I}Q_T \) and \( \mathcal{I}Q_P \) (see Section 14.2 for results).
Figure 14.3. A Snapshot of the XSnippet Interface - Result.

**Hypothesis 2:** Context-sensitive ranking heuristic provides better ranks for best-fit code snippets than context-independent heuristics. Moreover, context-independent heuristics degrade sharply with the increase in repository size (see Section 14.3 for results).

**Hypothesis 3:** Specialized queries, $\mathcal{I}_{Q_T}$ and $\mathcal{I}_{Q_P}$, combined with context-sensitive ranking heuristic provide better rank ordering for best-fit code snippets than generalized query ($\mathcal{I}_{Q_G}$) using context-sensitive ranking heuristic (see Section 14.4 for results).

**Hypothesis 4:** The XSnippet system provides significant assistance to developers, enabling them to efficiently complete a large variety of programming tasks (see Section 14.5 for results).
**Hypothesis 5:** The context-dependent approach of the *XSnippet* system allows
developers to complete more tasks than other previously proposed approaches (see
Section 14.6 for results).

In this section, we describe our experimental setup together with the results that
validate the above hypotheses.

### 14.1 The Experiment Setup and Methodology

The *XSnippet* system was deployed on a standalone PC Pentium 4 2.8 GHz
with 1 GB RAM running Microsoft Windows XP and Eclipse 3.1. The repository
used for the experiments contained approximately 2,000 Java class files and
22,000 methods extracted from two standard Eclipse plugins: `org.eclipse.jdt-.ui` and `org.eclipse.debug.ui`. These Java class files were transformed, with the
assistance of special purpose parsers, into their corresponding source code model
instances and subsequently loaded into the example repository.

To evaluate the *XSnippet* system, we designed 17 object-instantiation specific
programming tasks. This was a sufficiently large set to allow measurements of the
effects of different parameters, including the context of the code, sample availability,
difficulty level, and number of queries required to complete the tasks. All tasks were
based on the Eclipse plugin examples from *The Java(TM) Developer's Guide to
ECLIPSE (The 2nd Edition)* [9]. **Table 14.1** highlights the primary characteristics
of the tasks in terms of the type being queried ($t_q$), the parent ($CT_P$) and type
($CT_T$) contexts specified in the provided source class $C_s$, and the base type $t_b$
from which $t_q$ was directly (or indirectly) instantiated\(^1\) to complete programming tasks.

For each task, the declaration and usage of type $t_q$ together with all necessary Java

\(^1\) An object of type $t_q$ can be instantiated from one or more types $t_i,...,t_h$. These type $t_i,...,t_h$
may or may not exist in the code context. Moreover, the type $t_q$ can be instantiated from
the type $t_i$ in multiple ways. Each different instantiation may possibly have different behavioral effects
within the task.
source classes and jar files were provided. The object instantiation of type \( t_q \) was
left incomplete, requiring the completion of the code with the object instantiation
to successfully compile and execute the code.

<table>
<thead>
<tr>
<th>No.</th>
<th>( t_q )</th>
<th>( CF_p )</th>
<th>( CF_T )</th>
<th>( t_b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>IRunnable</td>
<td>ViewPart, ISelectionListener</td>
<td>Text, JavaMetrics, IWorkbenchPart, ISelection</td>
<td>ISelection</td>
</tr>
<tr>
<td>2.</td>
<td>IRunnable</td>
<td>ViewPart, ISelectionListener</td>
<td>Text, JavaMetrics, IWorkbenchPart, ISelection</td>
<td>ISelection</td>
</tr>
<tr>
<td>3.</td>
<td>IRunnable</td>
<td>ISessionChangedListener, ICompilationUnit, List, ElementChangedListener</td>
<td>ElementChangedEvent, Event</td>
<td>IEditorPart</td>
</tr>
<tr>
<td>4.</td>
<td>IRunnable</td>
<td>AddTraceStatementsAction, IEditorActionDelegate</td>
<td>IEditorPart, IAction</td>
<td>IEditorPart</td>
</tr>
<tr>
<td>5.</td>
<td>IRunnable</td>
<td>AddTraceStatementsAction, IEditorActionDelegate</td>
<td>IEditorPart, IAction</td>
<td>IEditorPart</td>
</tr>
<tr>
<td>6.</td>
<td>IRunnable</td>
<td>ViewPart, ISelectionService</td>
<td>Text, Composite</td>
<td>ViewPart</td>
</tr>
<tr>
<td>7.</td>
<td>IRunnable</td>
<td>TextEditorAction</td>
<td>-</td>
<td>TextEditorAction</td>
</tr>
<tr>
<td>8.</td>
<td>IRunnable</td>
<td>TextEditorAction</td>
<td>-</td>
<td>TextEditorAction</td>
</tr>
<tr>
<td>9.</td>
<td>IRunnable</td>
<td>TextEditorAction</td>
<td>ITextEditor</td>
<td>ITextEditor</td>
</tr>
<tr>
<td>10.</td>
<td>IRunnable</td>
<td>AbstractDecoratorTextEditor</td>
<td>SQLCodeScanner, AbstractSupport, Composite</td>
<td>TextEditor</td>
</tr>
<tr>
<td>11.</td>
<td>IRunnable</td>
<td>IEditorActionDelegate</td>
<td>IAction, TextEditor</td>
<td>TextEditor</td>
</tr>
<tr>
<td>12.</td>
<td>IRunnable</td>
<td>IEditorActionDelegate</td>
<td>IAction, IAction, IAction</td>
<td>TextEditor</td>
</tr>
<tr>
<td>13.</td>
<td>IRunnable</td>
<td>AddTraceStatementsAction, IEditorActionDelegate</td>
<td>IAction</td>
<td>JavaEditor</td>
</tr>
<tr>
<td>14.</td>
<td>IRunnable</td>
<td>IEditorActionDelegate</td>
<td>IAction</td>
<td>IEditorActionDelegate</td>
</tr>
<tr>
<td>15.</td>
<td>IRunnable</td>
<td>AddTraceStatementsAction, IEditorActionDelegate</td>
<td>IAction</td>
<td>IEditorActionDelegate</td>
</tr>
<tr>
<td>16.</td>
<td>IRunnable</td>
<td>IEditorActionDelegate</td>
<td>IAction</td>
<td>IEditorActionDelegate</td>
</tr>
<tr>
<td>17.</td>
<td>IRunnable</td>
<td>AddTraceStatementsAction, IEditorActionDelegate</td>
<td>IAction</td>
<td>JavaUI</td>
</tr>
</tbody>
</table>

| Manager | IEditorActionDelegate |

| Table 14.1. The Characteristics of 17 Programming Queries. |

It should be noted that the code snippets returned by the XSnippet system
varied in that (i) some code snippets could be seamlessly integrated with the code
under development, thereby enabling the task to be completed at once; (ii) some
code snippets introduced new objects into the context, requiring one or more new
queries to complete the task; and (iii) some code snippets encapsulated the type
\( t'_b \) that was either a generalization or specialization to the base type \( t_b \) exist in the
code context, requiring type modifications based on the type hierarchy to complete
the task. Code snippets in any one of the above three categories were considered
to be “desirable” as long as they allowed the task to be completed. We considered a task to be complete if it was both compilable and executable, and it enabled the required functionality.

14.2 Hypothesis 1. Effect of Query Types on Task Completion

In the first set of experiments, we measured the number of tasks out of the possible 17 tasks that could be completed using the code snippets returned by the XSnippet system. For this set of experiments, we varied the type of the instantiation queries from the generalized query $I_QG$ to the specialized queries – the type-based query $I _QT$ and the parent-based query $I _QP$. We measured the number of tasks that could be completed under each of the query types. For all of the experiments, the repository was set to contain all the source code model instances as detailed in Section 14.1.

Overall Performace. Figure 14.4 shows the percentage of tasks completed by the different types of queries $I _QG$, $I _QT$ and $I _QP$. The x-axis has the different query types and the y-axis is the percentage of tasks completed, with 100% representing all 17 tasks. The query $I _QG$ performed the best completing all 17 (100%) tasks, the query $I _QT$ completed 11 (65%) tasks, while $I _QP$ completed 12 (70%) tasks.

Categorization of Completed Tasks. Furthermore, the characteristics of the tasks supported by each of the query types were analyzed. Figure 14.5 summarizes these characteristics and relates them to the characteristics of the best-fit code snippets mined from the example repository. The left column shows the repository characteristics. The middle column shows the distribution of the tasks that adhere to the repository and code context characteristics. The tasks themselves are broken down into two groups – one where $t_b$ exists in the code context and one where $t_b$
Figure 14.4. The Percentage of Tasks Supported by the Different Types of Instantiation Queries - $\mathcal{IQ}_G$, $\mathcal{IQ}_T$, and $\mathcal{IQ}_P$.

The vertical lines on the right represent the results, in this case coverage of tasks by each query type.

The query $\mathcal{IQ}_T$ was able to support all tasks where (i) the provided source class $c_s$ encapsulated the base type $t_b$; and (ii) snippets in which $t_q$ was instantiated from $t_b$ ($t_b \rightarrow t_q$) were available in (mined from) the example repository. Twelve (12) tasks out of the total 17 tasks had the type $t_b$ in its code context. Out of these only 10 tasks had snippets where the condition $t_b \rightarrow t_q$ was true. The query $\mathcal{IQ}_T$ was also able to support tasks that required object instantiation via either a constructor or a static method invocation irrespective of the existence of base type $t_b$ in the code context. The 17th task required an object instantiation based on the static method invocation of the JavaUI class. This task was supported by the $\mathcal{IQ}_T$ query. In summary, the query $\mathcal{IQ}_T$ supported 11 tasks – 10 tasks requiring object instantiation of type $t_q$ based on the base type $t_b$ in the code context and
code snippets with \( t_b \rightarrow t_q \) available in the example repository, and 1 task requiring a static method invocation.

The query \( IQ_P \) supported all tasks where the parent context of the provided source class \( C_s \) matched the parent context of the class encapsulating the best-fit code snippet. The existence of the base type \( t_b \) in the code context, and the constraint \( t_b \rightarrow t_q \) had no impact on \( IQ_P \) performance. Twelve (12) tasks out of the possible 17 had a parent context match between the source class \( C_s \) and the class encapsulating the code snippets, and were completed by \( IQ_P \). Out of these 12 tasks, the base type \( t_b \) existed in the code context for only 10 tasks while the constraint \( t_b \rightarrow t_q \) held for only 8 of the tasks.
The query $I \mathcal{Q}_G$ supported all tasks. The constraint $t_b \rightarrow t_q$ and the existence of $t_b$ in the code context had no impact on the overall performance of $I \mathcal{Q}_G$. The results returned by $I \mathcal{Q}_G$ were a superset of the results returned by $I \mathcal{Q}_T$ and $I \mathcal{Q}_P$ ($S(I \mathcal{Q}_T) \subset S(I \mathcal{Q}_G)$ and $S(I \mathcal{Q}_P) \subset S(I \mathcal{Q}_G)$). The results of queries $I \mathcal{Q}_T$ and $I \mathcal{Q}_P$ had some overlap but were generally complimentary to each other.

**Number of Queries needed.** We found that in some cases a single query was not sufficient to complete a given task. This was primarily due to the fact that a code snippet selected from the results of the first query could introduce new object types not available in the given code context. Figure 14.6 shows the number of queries needed to complete the tasks under each query type $I \mathcal{Q}_T$, $I \mathcal{Q}_P$ and $I \mathcal{Q}_G$. The x-axis has the tasks given as integer number and the y-axis is the number of needed queries. All tasks supported by the query $I \mathcal{Q}_T$ were completed by a single query while some of the tasks completed by queries $I \mathcal{Q}_P$ and $I \mathcal{Q}_G$ required up to 3 queries. It is interesting to note that the tasks supported by both $I \mathcal{Q}_T$ and $I \mathcal{Q}_G$ required a single query. For the tasks supported by all queries, $I \mathcal{Q}_T$ and $I \mathcal{Q}_G$ had better overall performance with all 8 (100%) tasks completed using a single query, while $I \mathcal{Q}_P$ completed only 6 (75%) tasks using a single query and 2 (25%) tasks using two queries.

**Summary.** In summary, the generalized query, $I \mathcal{Q}_G$, provided better coverage of tasks than the specialized queries $I \mathcal{Q}_T$ and $I \mathcal{Q}_P$. Additionally, $I \mathcal{Q}_G$ and $I \mathcal{Q}_T$ had better overall performance than $I \mathcal{Q}_P$ with respect to the number of queries required to complete the tasks.
The Number of Needed Queries

**Figure 14.6.** The Number of Queries Needed to Complete the Tasks using Different Query Types $TQ_T$, $TQ_P$, $TQ_G$.

### 14.3 Hypothesis 2. Impact of Contextual Information on Ranking

We ran a second set of experiments to measure the effectiveness of the different ranking heuristics for different types of queries. Here, effectiveness is defined as the ability of the ranking heuristic to return the best-fit code snippet in the top-k results.

The best-fit code snippet for each of the 17 tasks was apriori determined and used to evaluate the different ranking heuristics – *length*, *frequency*, and *context* ranking heuristics, under each of query types $TQ_G$, $TQ_T$ and $TQ_P$. In addition, we used a *combination* of these ranking heuristics to hierarchically rank the results first by context then within a group by frequency and subsequently length. All experiments were conducted using the 17 tasks, the three query types and the complete repository set up as described in Section 14.1. We categorized the results of the effectiveness of the ranking heuristics based on each query type.
Context information and type-based query – $I_Q$. Figure 14.7 shows the cumulative distribution of the best-fit code snippet ranking for the different ranking heuristics. The results are reported for the 11 tasks supported by $I_Q$. The x-axis is the ranking given as integer numbers and the y-axis is the cumulative distribution. The combination ranking heuristic was by far the top performer, providing the best-fit code snippets for all 11 tasks ranked within the top-3. In fact, for 73% of the tasks the combination heuristic ranked the best-fit code snippet first. The context ranking heuristic performed similarly with the best-fit code snippets for all 11 tasks ranked within the top-4. The context heuristic ranked the best-fit code snippet first for 63% of its tasks (that is 63% of the tasks were completed by the first ranked code snippet). The frequency ranking heuristic performed poorly ranking the best-fit code snippets for all 11 tasks within the top-16. However, it should be noted that 54% of the tasks were completed by the first ranked code snippet. The length ranking heuristic performed the worst with the best-fit code snippets ranked within the top-35. The length heuristic while the worst overall, however, ranked the best-fit code snippet first for 63% of the tasks.

Context information and parent-based query – $I_P$. Figure 14.8 shows the cumulative distribution of the best-fit code snippet ranking for the different ranking heuristics. The results are reported for all 12 tasks supported by $I_P$. The combination ranking heuristic performed the best, ranking the best-fit code snippets for all 12 tasks within the top-25. For 33% of the tasks, the combination heuristic ranked the best-fit code snippet first. The context and frequency ranking heuristics performed similarly returning the best-fit code snippets for all 12 tasks ranked within the top-31. The context heuristic ranked the best-fit code snippet first for only 25% of the tasks. The frequency heuristic, on the other hand, ranked the best-fit code snippets first for 50% of the tasks (better than both combination and context heuristics). The length ranking heuristic performed the worst with the
best-fit code snippets ranked within the top-91. The length heuristic while the worst overall, however, ranked the best-fit code snippet first for 33% of the tasks, a result similar to the combination heuristic.

**Context information and generalized query – \( \mathcal{I}Q_G \).** Figure 14.9 shows the cumulative distribution of the best-fit code snippet ranking for the different ranking heuristics. The results are reported for all 17 tasks supported by \( \mathcal{I}Q_G \). The x-axis is the ranking given as integer numbers and the y-axis is the cumulative distribution. The combination ranking heuristic performed the best with the best-fit code snippets for all tasks ranked within the top-33. For 35% of the tasks, the combination heuristic ranked the best-fit code snippet first. The context ranking heuristic ranked the best-fit code snippet for all tasks within the top-80. For 35% of the tasks, however, the best-fit code snippets ranked first, a result similar to the combination

**Figure 14.7.** Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using \( \mathcal{I}Q_T \).
Figure 14.8. Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using $IQ_P$.

The Stability of Ranking Heuristics. Furthermore, the effect of the number of code snippets returned by each query type – $IQ_T$, $IQ_P$, and $IQ_G$, on the stability of the ranking heuristics was analyzed. Figures 14.10 - 14.12 show the ranking of the best-fit code snippets versus the number of result code snippets returned by $IQ_T$, $IQ_P$, and $IQ_G$, respectively. The x-axis is the number of code snippets returned by each query and the y-axis is the rank of the best-fit code snippet. For all query types,
the performance of the length and frequency heuristics degraded sharply for larger number of result snippets. The combination and context heuristics were relatively stable, even for large number of returned results. The performance difference was attributed to the sensitivity of the length and frequency heuristics to the size and richness of the example repository.

Summary. In summary, the combination ranking heuristic was the most effective ranking heuristic followed by the context and frequency ranking heuristics. The length ranking heuristic performed the worst for all query types.

14.4 Hypothesis 3. Effects of Query Types on Ranking

In the third set of experiments, we measured for a given task the ranks of the best-fit code snippets for different instantiation query types – $IQ_T$, $IQ_P$ and $IQ_G$. 

Figure 14.9. Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using $IQ_G$. 

```latex
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure14_9.png}
\caption{Cumulative Distribution of the Best-Fit Code Snippet Ranking for Different Ranking Heuristics using $IQ_G$.}
\end{figure}
\end{quote}

```
To level the playing field, all experiments were conducted using the 6 tasks supported by all three query types\(^2\). The ranking heuristic was fixed to the combination ranking heuristic for all queries and we used the complete repository setup as described in Section 14.1.

Table 14.2 shows the distribution of the ranks of the best-fit code snippets for the 6 tasks supported by the \(IQ_T\), \(IQ_P\) and \(IQ_G\) queries. Query \(IQ_T\) performed the best, having the best code snippet ranked 1\(^{st}\) 5 out of 6 times. Query \(IQ_P\) did the next best, with one fewer top-ranked snippets. Query \(IQ_G\) performed the worst returning the best-fit code snippet in the first rank only about 1/2 the time. This difference in performance was attributed to the fact that \(IQ_T\) and

\(^2\)Actually, there were 8 tasks supported by all three query types \(IQ_T\), \(IQ_P\) and \(IQ_G\). However, there were 2 tasks that neither the query \(IQ_T\) nor \(IQ_P\) could return the same desired snippets. These 2 tasks were thus discarded for consistency.
Figure 14.11. Variation in the Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $I_Q_P$.

<table>
<thead>
<tr>
<th></th>
<th>1&lt;sup&gt;st&lt;/sup&gt;</th>
<th>2&lt;sup&gt;nd&lt;/sup&gt;</th>
<th>3&lt;sup&gt;rd&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_Q_T$</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$I_Q_P$</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$I_Q_G$</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 14.2. Distribution of the Best-Fit Ranks for Different Query Types.

$I_Q_P$ filtered out non-candidate code snippets as part of the mining and snippet selection process while $I_Q_G$ returned all code snippets mined from the example repository. This effectively deteriorated the overall ranking for $I_Q_G$. In summary, specialized queries ($I_Q_T$ and $I_Q_P$) provide higher ranking best-fit code snippets when compared to the generalized query $I_Q_G$. 
Figure 14.12. Variation in the Rank for the Best-Fit Code Snippet with Increasing Code Snippet Results. Results reported for $IQ_G$.

14.5 Hypothesis 4. Analysis of XSnippet on Assisting Developers

In the fourth set of experiments, we analyzed the use of the XSnippet system in assisting developers to complete their programming tasks. Four tasks with varying degrees of difficulty were designed for the study. Table 14.3 gives a brief description of the four tasks together with a difficulty rating, as well as the ranking of the best-fit snippets returned by the system (For Task $C$, type modification based on type hierarchy must be applied to the best-fit code snippet to complete the task). Detailed descriptions of the tasks can be found in Appendix A. For each task, the declaration and usage of type $t_q$ together with all necessary Java source files, jar files and instructions on code execution were provided, but the object instantiation of the type $t_q$ was left incomplete. In all four tasks the provided code could not be compiled without completing the task modifications.
<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>( C^T_P )</th>
<th>( C^T_T )</th>
<th>Difficulty</th>
<th>Best-Fit Snippet Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Instantiate ISelectionService that tracks the selection within the package explorer</td>
<td>ViewPart, ISelectionListener</td>
<td>Text, Composite</td>
<td>Easy</td>
<td>1 or 2</td>
</tr>
<tr>
<td>B</td>
<td>Instantiate ICompilationUnit that represents the .java file selected from the package explorer</td>
<td>ViewPart, ISelectionListener</td>
<td>Text-Editor, IAction, IDocument</td>
<td>Easy-Medium</td>
<td>7 or 8</td>
</tr>
<tr>
<td>C</td>
<td>Instantiate TextSelection that represents the text selected from the Text editor</td>
<td>IEditorAction-Delegate</td>
<td>Hard</td>
<td>6*, 7* or 10*</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Instantiate ICompilationUnit that represents the .java file appearing on the Java editor</td>
<td>AddTraceStatements-Action, IEditorActionDelegate</td>
<td>IAction</td>
<td>Medium-Hard</td>
<td>27</td>
</tr>
</tbody>
</table>

**Table 14.3.** Brief Description of the Four Tasks Used for the User Study.

Volunteers from the UMass-Lowell population were solicited to participate in the study. Initially, one group of users was setup to complete the tasks without the use of XSnippet. These users were free to use the default Eclipse Java code assistant tool, the Eclipse API browser, search online via Google or use any other means available via the Internet. However, the users struggled for over one hour on the first task alone and, still unable to complete it, became frustrated and quit. It was therefore assumed for the majority of the users that they would be unable to complete the tasks without the use of XSnippet.

For the remainder of the study, each participant was trained to use the XSnippet system prior to them conducting the study. Participants were then randomly assigned to one of two possible groups. The first group was provided hints on the base type \( t_b \) from which an object of the query type \( t_q \) should be instantiated. These hints appeared as comments in the provided source code. The second group was not given any hints. For each participant, the number of tasks completed together with the development time to complete each task was recorded. In addition, the strategy employed to select the result snippet by each participant was monitored.

A special test harness was developed to conduct the user study. The test harness allowed participants to enter demographic and programming experience information.
at the start of the study as shown in Figure 14.13. For each task, the test harness provided a screen with a brief description of the task as shown in Figure 14.14, prior to launching the Eclipse development environment pre-set with all required files for the task. The test harness automatically recorded the time taken for the participants to complete each task. On completion of each task, participants were asked to fill a brief questionnaire as shown in Figure 14.15 that solicited information on the usefulness of the $XSnippet$ system.

All experiments were conducted using the 4 tasks, the generalized instantiation query $IQ_G$, the combination ranking heuristic, and the complete repository setup as described in Section 14.1.
Participant Statistics. Sixteen participants took part in the study. Out of these, seven participants were randomly binned into the hint group and nine participants were placed in the no hint group. Table 14.4 gives the participant characteristics. Here, most of the participants were Computer Science graduate students with one CS undergraduate in each group. The participants had comparable programming experience (C++ and Java) with most participants having on average about 1 year experience in one or both languages. Only 6 of the participants had used Eclipse as a development environment prior to conducting the study, and none of the participants had any experience with developing Eclipse plugins.

Given the participant population and distribution of programming experience, the hint group approximated the more knowledgeable and experienced developers, and the no hint group approximated the novice developers.
Figure 14.15. The Test Harness: The Questionnaire Screen.

**Tasks Completed.** For each participant the number of tasks completed was tracked. Figure 14.16 summarizes the results for task completion by participants in each group. The x-axis has the tasks and the y-axis is the percentage of participants that completed each of four tasks.

In the *hint* group, all participants completed Tasks A, B, and D. This 100% completion rate was helped in part by the provided *hints* – the participants used the provided hints to select the best-fit code snippet as opposed to utilizing the snippet ranking. For Task C, type modification in the code snippet was required to complete the task. Participants with experience and knowledge of Java type hierarchy (about 70% of participants) were able to complete the task. In the *no hint* group, all participants completed Tasks A and B, and 56% of the participants completed Tasks C and D. Participants in the *no hint* group were observed to use ranking
<table>
<thead>
<tr>
<th>Groups</th>
<th>Subjects</th>
<th>Level</th>
<th>C++</th>
<th>Java</th>
<th>Java Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>undergrad</td>
<td>6mo-1y</td>
<td>&lt; 3mo</td>
<td>NetBean</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>grad</td>
<td>1y-2y</td>
<td>&lt; 3mo</td>
<td>Others</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>3mo-6mo</td>
<td>NetBean</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>1y-2y</td>
<td>Others</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>&gt; 2y</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>grad</td>
<td>3mo-6mo</td>
<td>&gt; 2y</td>
<td>NetBean</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>&gt; 2y</td>
<td>Others</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>grad</td>
<td>&lt; 3mo</td>
<td>3mo-6mo</td>
<td>Others</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>grad</td>
<td>6mo-1y</td>
<td>3mo-6mo</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>6mo-1y</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>grad</td>
<td>6mo-1y</td>
<td>6mo-1y</td>
<td>Others</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>6mo-1y</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>grad</td>
<td>6mo-1y</td>
<td>1y-2y</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>undergrad</td>
<td>&lt; 3mo</td>
<td>1y-2y</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>grad</td>
<td>3mo-6mo</td>
<td>&gt; 2y</td>
<td>NetBean</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>grad</td>
<td>&gt; 2y</td>
<td>&gt; 2y</td>
<td>NetBean,Forte,Eclipse</td>
</tr>
</tbody>
</table>

Table 14.4. The Characteristics of Participants.

Figure 14.16. Percentage of Participants that Completed the Given Four Tasks in the Hint and No Hint Groups.
of the code snippets as their primary criteria for selection of snippets. For Task C participants had similar problems to the hint group with the type modification required in the code snippet. For Task D, the best-fit code snippet was ranked 27 accounting for the fewer number of participants that completed the task.

![Bar graph showing percentage of participants who completed tasks A, B, C, and D](image)

**Figure 14.17.** Percentage of Participants that Completed the Given Four Tasks in the Two Java Experience Classes: less than 1 year and greater than 1 year in Java Experience.

Furthermore, the relation of the participant’s Java experience to the completion of the tasks was analyzed. Participants were categorized into two experience classes, one with less than 1 year experience in Java and the other with greater than 1 year Java experience. Figure 14.17 shows the results of task completion by participants in each Java experience class. The x-axis has the tasks and the y-axis is the percentage of participants that completed each of four tasks. *XSnippet* was equally useful for participants in both groups for completing Tasks A, B and D. The notable difference was in the completion of Task C. Participants with more Java experience (> 1 year)
were all able to complete Task C, while only 25% of participants in the other group (< 1 year) were able to do so. This distinction was attributed to the knowledge of the Java type hierarchy in the more experienced Java participants.

**Task Development Time.** For each participant the development time for each task was tracked. The development time included (i) the Eclipse launch time; (ii) the query process time; (iii) the time taken for the participant to select a code snippet from the query results; and (iv) the time taken to modify and test the provided source code.

![Development Time Graph](image)

**Figure 14.18.** Development Time (Individual and Average) For Completing the Four Tasks. The Results are Categorized by the the *Hint* and *No Hint* Groups.

Figure 14.18 shows the time taken by each participant to complete the provided task. Development times for participants that did not complete the task were discarded. The x-axis has the tasks and the y-axis is the development time in minutes. The bars for each task represent the *hint* and *no hint* groups with the individual
participant times marked on the bars. The lines across the tasks represents the average time taken by the participants to complete each task in the *hint* and *no hint* groups.

Participants in both groups reported similar development times for completing Task A and Task B. For Task C and Task D, participants in the *hint* group fared better – selecting the best-fit code snippet based on the provided hints. The lower average development times for Task C and D for the *hint* group are representative of the improvement due to hints. Most participants in the *no hint* group tried several code snippets for Task C and Task D before hitting upon the best-fit code snippet. This was attributed to the fact that (i) for Task C, there were two or more code snippets that encapsulated the lexically visible types in the code context and were ranked higher than the best-fit code snippets. Additionally, type modification was required in the best-fit code snippets; and (ii) for Task D, code snippets returned by the query $\mathcal{Q}_G$ did not have context match with the provided source code.

**Selection Strategy.** For each participant the selection strategy for each task was tracked. Figure 14.19 shows the average number of attempts made by the participants (reflecting the number of times code snippets were selected) to complete the given 4 tasks. The x-axis has the ranks of best-fit code snippets and the y-axis is the average number of attempts made by the participants. The bars show 95% confidence interval around the mean of the number of attempts.

Participants took exactly 1 attempt to complete the given programming task when the best-fit code snippet was ranked first. When the best-fit code snippet was ranked below the top-10 results, participants averaged 2 attempts with a confidence interval (1.70, 3.17) to complete the task. With the best-fit code snippet ranked at 10, participant on average took 4 attempts with the confidence interval (2.26, 5.30), accounting for only 50% of participants completing the task. Similar results were reported for snippets ranked over the top-10 results. In fact, for the best-fit code
Figure 14.19. Average Number of Code Snippet Selections For Different Ranks of Best-fit Code Snippets.

snippet ranked above the top-10 results, there was no statistical difference between the number of attempts made for snippets that were ranked 10 or 27, indicating that it is critical to be able to rank the best-fit code snippets in the top-10 results.

14.6 Hypothesis 5. Head-to-Head Comparison

The last set of experiments was designed to compare the XSnippet system with other code assistant systems, namely Prospector [32] and Strathcona [23].

14.6.1 Prospector Versus XSnippet

XSnippet was directly compared to Prospector using the queries in Table 14.1. A direct comparison to Strathcona was not possible, as we were unable to obtain the Strathcona code base.
The repository for XSnippet was setup as described in Section 14.1. The default repository for the Prospector system was used. This repository contained complete source code from Eclipse 3.0 standard plugins, GEF plugins and Eclipse/GEF standard examples. In essence, the Prospector repository was a superset of the XSnippet repository. Moreover, as Prospector only reports the top 12 code snippets, the number of code snippets returned by XSnippet was also limited to 12 with the use of the combination ranking heuristic.

For this set of experiments, we varied the existence of base type \( t_b \) in the provided code, where \( t_b \) was the type from which \( t_q \) must be directly (or indirectly) instantiated to complete the task. The 17 tasks, shown earlier in Table 14.1, were used for comparison. For each task \( \mathcal{I}_Q \) and \( \mathcal{I}_G \) in the XSnippet system and \( \mathcal{I}_Q' \) in the Prospector system were used to return 12 results. Each result was inserted into the code, compiled and executed to check if the desired functionality was achieved. If the desired functionality was achieved for at least 1 of the 12 returned results, the task was counted as completed.

Figure 14.20 shows the percentage of tasks completed by \( \mathcal{I}_Q \), \( \mathcal{I}_Q' \) and \( \mathcal{I}_G \). The primary task characteristic used for analyzing the results was the existence (or lack of) of the base type \( t_b \) in the code context. The x-axis plots the two main criteria used for the comparison, and the y-axis plots the percentage of tasks completed by each query type. The XSnippet \( \mathcal{I}_Q \) performed better than Prospector's \( \mathcal{I}_Q' \) for tasks where the \( t_b \) existed in the code context. This performance difference can be attributed to the fact that Prospector query evaluation is limited to the lexically visible types within the class boundary where the superclass is discarded. The XSnippet \( \mathcal{I}_Q \) and Prospector's \( \mathcal{I}_Q' \) performed similarly for the tasks where \( t_b \) did not exist in the code context. The query \( \mathcal{I}_G \) performed the best and provided

---

\(^3\)\( \mathcal{I}_Q \) is an extension to \( \mathcal{I}_Q' \) by also taking the superclass type into account.
code snippets for 82% tasks irrespective of whether or not the base type \( t_b \) existed in the code context.

### 14.6.2 XSnippet Versus Prospector and Strathcona

XSnippet was also compared to Prospector and Strathcona using the queries that were reported in the evaluation study in [32] and [23] respectively. This evaluation, designed to measure the comparative performance of the XSnippet with Prospector and Strathcona, was conducted in two parts: comparison of Prospector with XSnippet, and comparison of Strathcona with XSnippet. The performance of the systems (XSnippet, Prospector, and Strathcona) was measured in terms of the number of tasks (queries) completed by the involved systems. If the system reported a result snippet that could be used for task completion, the task was considered to be complete. Ranking of the snippet was not taken into consideration.
For both parts of the experiment, $TQ_G$ was used as the query type of the XSnippet system and the complete XSnippet repository (see Section 14.1) was utilized. While ranking was not explicitly used, the ranking algorithm for XSnippet was set to combination ranking. The results reported for Prospector and Strathcona are directly taken from [32] and [23]$^4$ respectively.

**Comparison with Prospector.** Table 14.5 outlines the 20 queries used for the evaluation of the Prospector system. More detailed description of these queries can be found in [32]. These queries are defined based on the input type $t_b$ and the query type $t_q$. Thus, for example, query 1 requires the instantiation of the query type ImageDescriptor from the input type ImageRegistry.

We selected 15 (queries 5 through 18 and 20) out of the possible 20 queries for the comparative performance of Prospector and XSnippet. Queries 1 – 4 and 19 were pruned from the query set as either their input types or their query types were not available in the XSnippet repository$^5$.

Figure 14.21 shows the percentage of queries shown in Table 14.5 supported by Prospector and XSnippet. The x-axis has the two assistant systems – Prospector and XSnippet and the y-axis is the percentage of queries supported, with 100% representing 15 queries. The Prospector performed the best supporting all 14 (93%) queries while XSnippet supported 12 (80%) queries. The performance difference was attributed to the fact that the code snippets for the other 3 queries required one or more intermediate types not available in the XSnippet repository. For example, an instantiation of the query type MenuManager from the input type IViewPart required the intermediate type ApplicationWindow, IViewPart → Application-

---

$^4$We did not have access to Strathcona code base, and to keep the comparison uniform we applied the same methodology to the comparison with Prospector.

$^5$In the current version the XSnippet repository contains only class files extracted from two Eclipse standard plugins.
Table 14.5. The Characteristics of 14 Programming Queries Reported in Prospector [32].

<table>
<thead>
<tr>
<th>No.</th>
<th>Programming problem</th>
<th>$t_b$</th>
<th>$t_q$</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Get image handle for lazy image loading</td>
<td>ImageRegistry</td>
<td>ImageDescriptor</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Open a named file for memory-mapped I/O</td>
<td>String</td>
<td>MappedByteBuffer</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Get table widget from an Eclipse view</td>
<td>TableViewer</td>
<td>Table</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Retrieve canvas from scrolling viewer</td>
<td>ScrollingGraphical-</td>
<td>FigureCanvas</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Read lines from an input stream</td>
<td>InputStream</td>
<td>BufferedReader</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Get the active editor</td>
<td>IWorkbench</td>
<td>IEditorPart</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Get window for Message Box</td>
<td>KeyEvent</td>
<td>Shell</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Convert legacy class</td>
<td>Enumeration</td>
<td>Iterator</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Get selection from event</td>
<td>SelectionChangedEvent</td>
<td>ISelection</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Iterator over map values</td>
<td>Map</td>
<td>Iterator</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Add menu bars to a view</td>
<td>IViewPart</td>
<td>MenuManager</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Set captions on table columns</td>
<td>TableViewer</td>
<td>TableColumn</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Track selection changes in another widget</td>
<td>IEditorSite</td>
<td>ISelectionService</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Read lines from a file</td>
<td>String</td>
<td>BufferedReader</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>Find out what object is selected</td>
<td>IWorkbenchPage</td>
<td>IStructuredSelection</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>Manipulate document of visual editor</td>
<td>IWorkbenchPage</td>
<td>IDocumentProvider</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>Convert file handle to file name</td>
<td>IFile</td>
<td>String</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>Get an Eclipse view by name</td>
<td>IWorkspace</td>
<td>IViewPart</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>Set graph edge routing algorithm</td>
<td>AbstractGraphical-</td>
<td>ConnectionLayer</td>
<td>no</td>
</tr>
<tr>
<td>20</td>
<td>Retrieve file from workspace</td>
<td>IWorkspace</td>
<td>IFile</td>
<td>no</td>
</tr>
</tbody>
</table>

**Window $\rightarrow$ MenuManager.** While both MenuManager and IViewPart existed in the XSnippet repository, the ApplicationWindow did not exist. XSnippet was thus not able to return the code snippet. However, it should be noted that if the MenuManager did exist in the XSnippet repository, then the code snippet IViewPart $\rightarrow$ ApplicationWindow $\rightarrow$ MenuManager would be retrieved using a single query or otherwise using two queries results in ApplicationWindow $\rightarrow$ MenuManager and IViewPart $\rightarrow$ ApplicationWindow. In addition, XSnippet was able to find desired code snippet IWorkspace $\rightarrow$ IFile while Prospector had a limitation for retrieving the desired one.

**Comparison with Strathcona.** Figure 14.22 outlines the 4 queries used for the evaluation of the Strathcona system. More detailed description of these queries can be found in [23]. These queries are defined based on the structural context descriptions encapsulated in the code being queried at the granularity of a method.
- types, calls and parents. Thus, for example, query 1 requests for code samples having similar types – `IStatusLineManager`, `SourceViewer` and `Action`, calls – `setMessage(String)` defined in `IStatusLineManager`, and parents – `ViewPart`.

We selected 2 (queries 1 - 2) out of the possible 4 queries for the comparative performance of Strathcona and XSnippet. Queries 3 – 4 were pruned from the query set as they are not applicable for object instantiation and their results are not reported in [23].

Figure 14.23 shows the percentage of queries shown in Table 14.22 supported by Strathcona and XSnippet. The x-axis has the two assistant systems – Strathcona and XSnippet and the y-axis is the percentage of queries supported, with 100% representing 2 queries. Both Strathcona and XSnippet supported all 2 (100%) queries.
public class CodeViewer extends ViewPart {
    private SourceViewer aViewer;
    private Action copyAction, selectAllAction;
    private Action createASTAction, pasteAction;

    // 1. Update status line
    private void updateStatusBar(String msg) {
        IStatusLineManager.setMessage(msg);
    }

    // 2. Create AST
    private void createASTFromSource(String source) {
        ASTParser.setSource(source.toCharArray());
    }

    // 3. Highlight Text
    private void highlightRegions(Vector regions) {
        StyleRange[] srs = new StyleRange[regions.size()];
        aViewer.getTextWidget().setStyleRanges(srs);
    }

    // 4. Generate method signature
    public boolean visit(MethodDeclaration node) {
        node.getModifiers();
        node.getName().getIdentifier();
        node.parameters();
        return super.visit(node);
    }

    Figure 14.22. The Characteristics of 2 Programming Queries Reported in Strathcona [23].
Figure 14.23. Percentage of Queries Supported By Strathcona and XSnippet.
CHAPTER 15
RELATED WORK

Many approaches for mining code samples [23, 32, 22, 8, 36] have been proposed in literature to assist developers when implementing software systems. We classify these approaches into two categories: \textit{context-sensitive mining} [23, 32, 22]; and \textit{context-independent mining} [8, 36].

15.1 Context-sensitive Mining

Holmes et al. [23] have developed Strathcona, an Eclipse plug-in, that enables location of relevant code in an example repository. Their approach is based on heuristics that match the structural context descriptions encapsulated in the code under development with that of the example code. The structural context descriptions are (i) the types declared and used in a method in which a query is initiated; (ii) the calls (invocations) used in a method in which a query is initiated; and (iii) the direct parents (superclasses and interfaces) as well as the types of the class fields. Based on this categorization of structural context descriptions, they define six structural matching heuristics: inheritance, basic calls, calls best fit, calls with inheritance, basic uses, and uses with inheritance. The result is a set of examples that occur most frequently when applying all heuristics. Each example result consists of a code snippet, a structural description of the code snippet and a rationale explaining the relevance of the code snippet to the problem the developer is facing. The approach presented by Holmes et al, while a good step forward, has some drawbacks. First, each heuristic is \textit{not refined} with respect to a particular task, resulting
often in too many irrelevant hits. For example, consider the object instantiation of a query type \( t_q \) as a query. Strathcona results may include (together with the desired results) a set of code snippets that (a) only use an object of the query type \( t_q \) either via the invocations of its methods or as a method parameter. This is because there is no distinction between instantiation and usage of a type \( t_q \) in the heuristics; or (b) refer to types other than the query type \( t_q \). This is due to the fact that all types are treated equally by the heuristics. Second, Strathcona heuristics are sometimes overly constrained, resulting in too few hits. For example, the basic uses heuristic in Strathcona utilizes all types declared and used in the method being queried, and returns methods that use the same types. However, it is seldom that all types are related. Hence, it is likely that the basic uses heuristic may filter out many potential code snippets.

Mandelin et al. [32] have developed Prospector, a tool integrated with the Eclipse IDE code assistance feature. Prospector provides the support for the retrieval of code snippets that instantiate an object of a query output type \( \mathcal{T}_{out} \) from a given input type \( \mathcal{T}_{in} \). Prospector infers queries from the context of a developer’s code. Specifically, a set of queries, denoted as \((\mathcal{T}_{in1}, \mathcal{T}_{out}), (\mathcal{T}_{in2}, \mathcal{T}_{out}), \ldots, (\mathcal{T}_{in_n}, \mathcal{T}_{out})\), are created using the input types of lexically visible objects. In addition, \texttt{void} is used as an input type if none of visible objects can be used to instantiate an object of the query type. That is an object is instantiated via a constructor or a static method. This approach is based on the Jungloid graph that is created using both API method signatures and a corpus of sample client programs – Eclipse 3.0 standard plugins, GEF plugins and Eclipse/GEF standard examples, and consists of chains of objects connected via method calls. The code snippet retrieval is thus accomplished by traversing a set of paths encapsulated in the Jungloid graph from \( \mathcal{T}_{in} \) to \( \mathcal{T}_{out} \) wherein each path represents a code fragment and a set of paths in turn composes all code fragments to form a code snippet. As there can be one
or more sets of paths, the code snippets are ranked using the length of the paths with the shortest path ranked first. While Prospector provides more refined, objectinstantiation specific queries than Strathcona, it still returns many irrelevant hits and in some cases too few qualified hits. This is mainly because of (i) overuse of API method signature. For example, consider the partial signature graph from AST to IDocument in Figure 15.1 and its corresponding set of partial snippets retrieved using the query (AST,IDocument) shown in Figure 15.2. Here, List connects a path from AST to IDocument. However, the type of the elements in the List generated from the method structuralPropertiesForType() of ASTNode class is different from DocumentEvent – an element type in the List generated using the method mergeProcessedDocumentEvents() of TextUtilities class. This code snippet while compilable causes a runtime error; and (ii) the context description is limited to only the visible input types declared in the boundary of the method and the class ignoring the “parent” and thereby missing a set of potentially qualified hits. For example, consider the instantiation for an object iss of type ISelectionService in the method createPartControl illustrated in Figure 10.3. Here, the result getViewportPart().getWorkbenchWindow().getSelectionService() is not returned as the parent context, viewport, is not taken into account by Prospector.

![Diagram](image)

**Figure 15.1.** The Partial Signature Graph from AST to IDocument.
1. ASTNode aSTNode = ast.newWildcardType();
   List list = aSTNode.structuralPropertiesForType();
   DocumentEvent documentEvent = TextUtilities.mergeProcessedDocumentEvents(list);
   IDocument document = documentEvent.getDocument();

2. ASTNode aSTNode = ast.newCompilationUnit();
   List list = aSTNode.structuralPropertiesForType();
   DocumentEvent documentEvent = TextUtilities.mergeProcessedDocumentEvents(list);
   IDocument document = documentEvent.getDocument();

3. ASTNode aSTNode = ast.newPackageDeclaration();
   List list = aSTNode.structuralPropertiesForType();
   DocumentEvent documentEvent = TextUtilities.mergeProcessedDocumentEvents(list);
   IDocument document = documentEvent.getDocument();

Figure 15.2. The Prospector’s Partial Snippet Results where AST and IDocument is an input and output type.

Hill et al. [22] have developed a method completion tool as a plugin for jEdit 4.2. The tool automatically completes a method body based on the current context of a method being developed using machine learning technique. Their approach defines each method as a 154-dimension vector containing the number of lines of code, the number of arguments, a hash of the return type, the cyclomatic complexity, and the frequency count of each of the 150 Java Language token types. The vector $v_q$ of a method being queried is compared to a set of pre-computed vectors $v_t$ of methods in the example repository, and the best method match is returned to the developer where the best match is defined by the smallest difference between $v_q$ and $v_t$. This enables the developer to view similar methods and to subsequently choose the best-fit method to complete the method under development. This approach suffers from some of the same drawbacks as Strathcona – the approach is general to all types of queries and cannot be used to ask a more directed query. Moreover, the approach is inflexible using a complete vector comparison at all times, and potentially missing out on partial matches that may be of use. Like Prospector, this approach limits the context to just the type context and does not make use of the parent context.
15.2 Context-independent Mining

Cubranic et al. [8] have developed Hipikat that forms an implicit group memory from the information stored in a project’s archives, and recommends software artifacts from its archives that are relevant to a programming task a developer is trying to perform. The implicit group memories are formed when software artifacts are added to a project’s history. There are four types of artifacts: bug and feature descriptions (e.g. items in Bugzilla), source code revisions (e.g. checked in a CVS source repository), messages posted on developer forums (e.g. newsgroup and mailing lists), and other project documents (e.g. design documents). To form a group memory, these artifacts are connected to each other by different types of relationships. For example, the task description $T_1$ is linked to the task description $T_2$ via a similar_to relationship if the task $T_1$ and $T_2$ are similar. The source file $S_1$ is linked to the task description $T_1$ via an implement relationship if $S_1$ is the implementation of $T_1$. The group memory is used to direct the selection of relevant artifacts in response to a query. For example, once a developer has started working on a task, the developer may be interested in other tasks that have been completed within the same subsystem, or with a similar description. Following the similar_to relationship may lead to tasks that are helpful. Once a similar task has been identified, following the implement relationship can lead to the source revision that implements the task of interest. This approach supports new developers joining an existing software development team by allowing them to gain knowledge about the intricacies of the system, the development processes being used, as well as the organizational structure surrounding the project. However, this approach relies solely on the in-house development history [23], thereby limiting its applicability for retrieving other artifacts that may potentially be useful for the task in hand. Moreover, the queries are typically specified in terms of the broad task descriptions, making the queries
themselves too generic for producing meaningful answers applicable to a particular task.

Michail et al. [37, 35, 36] have developed CodeWeb – tool to analyze a large collections of applications and hence the characteristic usage of the library. Specifically, the CodeWeb tool uses data mining techniques to discover “reuse patterns” that can be employed to guide and check the library usage. The reuse patterns are if/then rules which indicate that application classes that contain the antecedent class tend to also contain the consequent class. Given the reuse pattern chosen by a developer, CodeWeb provides direct access to the application source code for examples of library usage. This approach supports developers who are not familiar with the domain and may not know what components in the library that are relevant. This approach is complimentary to the XSnippet approach, providing a first step in the tool set required for assisting developers.
CHAPTER 16

SUMMARY

We present XSnippet, an Eclipse plugin that allows developers to query for relevant code snippets from a sample code repository. Here, relevance is defined by the context of the code, both in terms of the parents of the class under development as well as lexically visible types. Figure 16.1 highlights the primary contributions of this work. Queries, invoked from the Java editor, can range from the generalized object instantiation query that returns all possible code snippets for the instantiation of a type, to the more specialized object instantiation queries that return either parent-relevant or type-relevant results. Queries are passed on to a graph-based Snippet Mining module that mines for paths that meet the requirement of the specified query. Paths here can be either within the method scope or outside of the method boundaries, ensuring that relevant code snippets that are spread across methods are discovered. All selected snippets are passed on to the Ranking module that supports four types of ranking heuristics: context-sensitive ranking, frequency-based ranking, length-based ranking, and a ranking heuristic that combines the three.

Our experimental evaluation of the XSnippet system has shown: 1) a generalized query provides better coverage of tasks than does a specialized query. In our experiments, the generalized query $I_Q G$ was able to cover (provide best-fit snippets) for all tasks; 2) a specialized query provides better-fit code samples at a higher rank when used with a context-sensitive ranking algorithm; 3) context-sensitive ranking performs significantly better than context-independent ranking and is less suscep-
Figure 16.1. An Overview of the XSnippet Framework.

tible to variations in the size of the repository; 4) XSnippet has the potential to assist developers by providing sample code snippets relevant to their task at hand, and to help decrease the overall development time; and 5) XSnippet provides better coverage of tasks and better ranking for best-fit snippets than other code assistant systems.
Part IV

Conclusions and Future Work
CHAPTER 17

CONCLUSIONS AND FUTURE WORK

Software reuse has been targeted as a panacea for reducing software costs and for improving both software quality and programmer productivity. Today, it is accepted practice for software developers to reuse software units, that range from *ascii* units such as code fragments, to *binary* units such as functions, modules and even entire application systems, to both guide and reduce their development efforts. These software units are typically bundled with library or framework packages, are available as in-house and off-the-shelf components, or are accessible via books, manuals, web sites, live application code and code that the developer has written previously. Software unit procurement is the first key step of the reuse process at the software implementation stage, and is often broken down into *component procurement* and *code sample procurement* based on the two popular types of software units. While tools exist to assist developers in this procurement process, the tools are limited in their capabilities by the type of queries they provide, the efficient matching/mining techniques that underlie the procurement process and the effective ranking techniques used to evaluate the procured units. We broadly classify these limitations as *quality* of procured units and the effective *ranking* of the same.

17.1 Contributions of this Dissertation

17.1.1 Component Procurement

In Part II of this dissertation, we present *XCoDE*, an XML-based Component Discovery and Evaluation framework, that allows developers to query a compo-
nt repository for components based on the given query component interface. In particular, XCoDE makes the following contributions:

- **Description Model:** XCoDE presents an XML-based unifying component description model (XCM) [58] that serves as a virtual schema for components conforming to different component models. Specifically, XCM provides a standard for the definition of components that crosscuts the different component models and unifies the variance between different models, aiding both component retrieval and evaluation.

- **Matching Algorithm:** XCoDE provides an innovative hybrid match algorithm, QoMym [59], to retrieve a set of candidate components with respect to a query component interface. The QoMym algorithm combines different matching algorithms – type matching algorithm to determine the match between two given domain types or signatures, and linguistic matching algorithm to define the match between two given labels. Specifically, these matching algorithms are combined based on the significant value of each information contained in a component. For example, for a method, the label typically has more significance than the signature.

- **Match Metric:** XCoDE supports a unique Quality of Match (QoM) metric [57, 59] that measures the “goodness” of a match. QoM is based on a match taxonomy and a weight-based match model that qualitatively and quantitatively evaluate a match between a query and a library component. These are based on information ranging from the type hierarchy to the labels of properties, methods and events of the two components. The combination of the match taxonomy and the weight-based match model provides a clear distinction between two or more matches.
We present a set of experiments to validate our approach and evaluate the benefits of XCoDE. In particular, we measure the match quality using the precision and recall used heavily in the information retrieval domain. Head-to-head comparisons have been made between XCoDE and other state-of-the-art algorithms found in literature, namely signature, linguistic and a filtering algorithms. Our experimental evaluation has shown that with the combined use of the semantic and syntactic information we are able to achieve higher precision and recall without the performance overhead associated with approaches like specification matching. Moreover, our results suggest that a disciplined combination of different algorithms (linguistic and signature) can provide better overall quality than a naive filter-based approach.

17.1.2 Code Sample Procurement

In Part III of this dissertation, we present XSnippet, an XML-based Code Snippet Mining and Ranking (XSnippet) framework, that allows developers to query a sample repository for code snippets relevant to the programming task at hand. In particular, XSnippet makes the following contributions [60]:

- Range of Queries: XSnippet supports a range of object-instantiation specific queries from specialized to generalized. These queries allows developers to switch between a context-independent retrieval of code snippets to various degrees of context-sensitive retrieval.

- Source Code Model: XSnippet presents a graph-based source code model that captures the structure and behavior of a source class and provides a formal model for the code snippet mining. Both structure and behavior are a rich syntactic and semantic resource that enables not only the provision of the code context to facilitate the ranking, but also supports newly defined queries.

- Mining Algorithm: XSnippet provides an innovative code mining algorithm, DFSMINE, that accommodates the range of object-instantiation specific queries
by constraining the mining process as needed. Specifically, the DFSMINE algorithm is based on a graph representation of Java source class and mines code snippets that exist either within the scope of an individual method or that are spread across the boundaries of two or more methods.

- Ranking Heuristics: \textit{XSnippet} supports a set of ranking heuristics from context sensitive to context independent. The context sensitive heuristic ranks a set of code snippets based on the contextual information harnessed from the code being queried, while the context independent heuristics rank code snippets based either on the snippet length or the number of times identical code snippets written within or across source classes mined from the retrieval process.

We present a set of experiments to validate our approach and to evaluate the potential benefits of \textit{XSnippet}. In particular, tradeoffs have been studied, examining coverage and ranking for different types of queries ranging from generalized to specialized queries, and the usability of the system via a user study. Head-to-head comparisons have been made between \textit{XSnippet} and the \textit{Prospector} code assistant system [32]. Our experimental evaluation has shown that \textit{XSnippet} has significant potential to assist developers, and provides better coverage of tasks and better rankings for best-fit code snippets than other code assistant systems.

17.2 Future Work

17.2.1 Queries

For the usability of \textit{XSnippet}, it is essential we need to provide a set of queries that covers all potential programming hurdles faced by developers when implementing software systems. In \textit{XSnippet}, we have covered object instantiation queries but have not addressed queries such as method invocation or explored other types
of queries. An interesting future work would be to extend the queries beyond the object instantiation to queries that pertain to the correct usage of methods. These correspond to discovering embedded sequences of methods that must be used together to ensure proper method behavior such as: 1) methods linked by the life cycle of their objects. For example, to ensure correct behavior of an object of type IWorkingCopyManager the methods connect, getWorkingCopy and disconnect must be used together; and 2) methods commonly interconnected by an invocation path. For example, method apply of the TextEdit class is commonly used in concert with the method rewriteAST of the class ASTRewrite to apply the edit-tree returned by the method rewriteAST() to the given document.

17.2.2 Virtual Repository

Web search engines such as Google [6] and Yahoo store full HTML of every web page from heterogeneous repositories in their local repositories, enabling a single point access to different repositories and supporting a large set of keyword-based queries. To the best of our knowledge, no previous work has addressed the development of a virtual repository as a central repository for different software unit repositories to facilitate the discovery of qualified software units across heterogeneous repositories. In this dissertation, our repositories contains a medium set of components and code samples particular for validating our approach. An interesting future work would be the development of crawling techniques that enable the automatic detection and subsequent information extraction of components and code samples that are typically bundled with library or framework packages, are available as in-house and off-the-shelf components, or are accessible via web sites, real-life application code and code that the developer has written previously. This enables the completeness of the repository while providing a single point access to different repositories, thereby supporting various developer needs in procuring software units.
17.2.3 Optimization of Search Space

Performance of a match algorithm is largely dependent on the search space that must be examined to determine all potential candidate software units. In the component procurement, at the lowest level an exhaustive pairwise comparison between the methods (or properties and events) of a query component and a set of library components is conducted [68, 69, 26, 17]. In cases when a large number of pairs are unrelated, this is an expensive and unnecessary process. At the component level, components in the repository are structured based solely on their classification schemes or clustered based on the most general components [26], while other essential issues such as domain and indexing are not taken into account [17]. It is thus likely that potential non-candidate components would be involved in the retrieval process, making the retrieval process time-consuming and expensive. The current code sample procurement suffers from similar drawbacks as no auxiliary structures are used to organize code samples in the repository.

An interesting future work in optimizing component procurement would be to develop an adaptable classification scheme to cluster components as a whole as well as individually based on component information. For example, methods of a library component can be clustered based on the method labels. Each method of a query component can then be compared to a group of library methods instead of all library methods. Another venue for optimization is the application of software domains proposed by Sugumaran et al. [56] and the combination of different techniques for structuring repositories.

In the XSnippet framework, we apply indexing techniques that is commonly used in the database application domain to index the code samples to facilitate the code sample procurement. The index is based solely on domain types declared or referred to in source classes particular to the purpose of facilitating object-instantiation specific queries. Another venue for optimization that we have not examined is the
use of the reuse patterns introduced by Michail et al. [37, 35, 36]. The reuse patterns are the \textit{if/then} rules indicating the source class that contain the antecedent class tend to also contain the consequent class. Here, we can index code samples in the repository based on two domain types, $t_1$ and $t_2$, where $t_1$ is initialized in the source class and $t_2$ is the type that contains $t_1$ in such source class. This would further enable the reduction of the search space especially for querying an instantiation of type $t_1$ with the type context $t_2$. 
APPENDIX A

TASKS IN USER STUDY
A.1 Programming Task A

Overview. Implement an Eclipse plugin that extends the JDT user interface by adding a new view JDG2E: Java Metrics. This view shows the message *You have successfully accomplished this programming task!* for any selection within the package explorer. All necessary Java source files and jar files involved with this Eclipse plugin are provided.

Task. Instantiate an object `iss` of type `ISelectionService` that tracks the selection within an object, the package explorer. This object is located in the method `createPartControl()` of `JavaMetricsView.java` (defined in the plugin project `ProgrammingTaskA` and package `com.ibm.jdg2e.jdt`) as shown in Figure A.1.

```java
public class JavaMetricsView extends ViewPart implements ISelectionListener, IJavaMetricsListener {
    Text message;

    public void createPartControl (Composite parent) {
        parent.setLayout(new FillLayout());
        message = new Text(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
        ISelectionService iss;
        iss.addSelectionListener (this);
    }
}
```

**Figure A.1.** The Partial `JavaMetricsView` Class.

Test. Select Run > Run As > Eclipse Application to open the new Eclipse workspace (Make sure that the plugin project `ProgrammingTask1` is selected). Single click to select anything within the package explorer. You should see the message *You have successfully accomplished this programming task!* in the view JDG2E: Java Metrics as shown in Figure A.2. (To open the view JDG2E: Java Metrics, select Window > Show view > Other > Java > JDG2E: Java Metrics.)
single click to select anything within the package explorer

**Figure A.2.** The Expectation of Programming Task A.
A.2 Programming Task B

Overview. Implement an Eclipse plugin that extends the JDT user interface by adding a new view JDG2E: Java Metrics. This view shows the number of methods, fields and literals encapsulated in the .java source file that is selected within the package explorer. All necessary Java source files and jar files involved with this Eclipse plugin are provided.

Task. Instantiate an object cu of type ICompilationUnit that represents an entire .java source file. This object is located in the method selectionChanged() of JavaMetricsView.java (defined in the plugin project ProgrammingTaskB and package com.ibm.jdg2e.jdt) as shown in Figure A.3.

```java
public class JavaMetricsView extends ViewPart implements ISelectionListener, IJavaMetricsListener {
    Text message;
    public void createPartControl (Composite parent) {
    }
    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        ICompilationUnit cu;
        jm.reset(cu);
    }
}
```

Figure A.3. The Partial JavaMetricsView Class.

Test. Select Run > Run As > Eclipse Application to open the new Eclipse workspace (Make sure that the plugin project ProgrammingTask2 is selected). Single click to select any .java source file in the package explorer. You should see the number of methods, fields and literals defined in the selected .java source file in the view JDG2E: Java Metrics as shown in Figure A.4. (To open the view JDG2E: Java Metrics, select Window > Show view > Other > Java > JDG2E: Java Metrics.)

178
Figure A.4. The Expectation of Programming Task B.
A.3 Programming Task C

Overview. Implement an Eclipse plugin that extends the JDT user interface by adding a new action to the text editor. This plugin contributes a new **Word Count** toolbar button that counts and shows the number of selected words – selected within the text editor – in the dialog window.

Task. Instantiate an object `ts` of type `ITextSelection` that represents a textual selection. This object is located in the method `run()` of `TextEditorWordCountAction.java` (defined in the plugin project ProgrammingTaskC and package com.-ibm.jdg2e.actions.ide) as shown in Figure A.5.

```java
public class TextEditorWordCountAction implements IEditorActionDelegate {
    TextEditor textEditor;
    
    public void run(IAction action) {
        IDocument document = textEditor.getDocumentProvider().getDocument(textEditor.getEditorInput());
        ITextSelection ts;
        try {
            String text = document.get(ts.getOffset(), ts.getLength());
        } ............
    }
}
```

**Figure A.5.** The Partial TextEditorWordCountAction Class.

Test. Select **Run > Run As > Eclipse Application** to open the new Eclipse workspace (Make sure that the plugin project ProgrammingTask3 is selected). Open (double click) the .txt file in the package explorer such that it appears on the text editor, highlight any number of word on the text editor, and click the **Word Count** toolbar button. You should see the dialog window that displays the number of selected words as shown in Figure A.6.
Figure A.6. The Expectation of Programming Task C.
A.4 Programming Task D

Overview. Implement an Eclipse plugin that extends the JDT user interface by adding a new action to the Java editor. This plugin contributes a new Add Trace Statement toolbar button to analyze the .java source file that is currently shown in the Java editor, and insert trace statements that output the name and parameters of each method.

Task. Instantiate an object cu of type ICompilationUnit that represents an entire .java source file. This object is located in the method run() of AddTraceStatementsEditor.java (defined in the plugin project ProgrammingTaskD and package com.ibm.jdgt2e.jdt) as shown in Figure A.7.

```java
public class AddTraceStatementsEditorAction extends AddTraceStatementsAction
    implements IEditorActionDelegate {

    public void run(IAction action) {
        ICompilationUnit cu;
        List methodDeclarations = new JavaMethodCollector(cu).getMethodDeclarations();
    }
}
```

Figure A.7. The Partial AddTraceStatementsEditor Class.

Test. Select Run > Run As > Eclipse Application to open the new Eclipse workspace (Make sure that the plugin project ProgrammingTask4 is selected). Open (double click) any .java file such that it appears on the Java editor and then click Add Trace Statements toolbar button. You should see the statements that output the name and parameters of each method as shown in Figure A.8.
Figure A.8. The Expectation of Programming Task D.
BIBLIOGRAPHY


