AN EFFECTIVE CODE SEARCH TECHNIQUE FOR RETRIEVING FUNCTIONALLY SIMILAR METHODS

ABDUS SATTER
MSSE 0401

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

AN EFFECTIVE CODE SEARCH TECHNIQUE FOR RETRIEVING FUNCTIONALLY SIMILAR METHODS

ABDUS SATTER

Approved:

Signature *Date*

_____ _____

Student: Abdus Satter

_____ _____

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *Md. Akteruzzaman and Rajia Sultana*, my parents
who have always been there for me and inspired me

Abstract

During software development, code search is commonly performed to find reusable code snippets. However, the effectiveness of a code search engine depends on the retrieval of all the functionally similar code snippets. Usually, code search engines extract textual keywords from the code fragments to construct the index. It is very common that functionally similar codes contain different keywords. So, the code snippets are indexed against different terms. When a user query matches to any of the index terms, only the corresponding code fragments will be retrieved.

In this thesis, a technique named Similarity Based Method Finder (SBMF) is proposed to index functionally similar methods against common relevant terms. Two or more methods are considered as functionally similar if the methods produce the same outputs for the same input set. All the functionally similar methods need to be indexed against common relevant terms. It will assist to retrieve semantically similar methods all together. SBMF checks functional similarity among codes by executing and matching outputs against the same set of inputs. Next, it determines relevant index terms by finding keywords that are found in most of the code snippets. To increase the matching probability between query terms and index terms, user query is expanded by adding topic and context specific keywords.

In order to evaluate the query expansion process of SBMF, an experiment was conducted on 22 real life user queries. The experimental analysis shows that SBMF produces 48.6% more precision@10 than an existing technique named Thesaurus-

Based Automatic Query Expansion. Another experiment was performed on 50 open source projects to evaluate the retrieval effectiveness of SBMF. According to the experimental results, on an average, SBMF retrieves 38% and 58% more relevant methods than two existing techniques named Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS), respectively. Moreover, SBMF retrieves self executable code snippets which are easily pluggable in the intended development context, and thus reduces time and effort while reusing code.

Acknowledgments

"All praises are due to Allah"

First of all, I praise Allah, The Almighty and The Lord of The World, for giving me this opportunity and granting me the ability to continue my research work effectively.

I would like to thank my supervisor Professor Dr. Kazi Muheymin-Us-Sakib for his support and guidance during my candidature. His constant encouragement was one of the main driving forces that kept me motivated. I would like to thank him for all his help with the research directions and experimentation. All his comments and suggestions were invaluable. His very demanding nature of supervising as well as pushing me to the limits have helped me grow as a researcher.

I am grateful to Dr. Md. Saidur Rahman and Dr. Md Abdur Razzaque for their excellent feedbacks and suggestions to enhance this work further.

I would like to thank my committee members for serving even at hardship. I also want to thank them for letting my defense be an enjoyable moment and for their brilliant comments and suggestions.

Also I would like to thank other faculty members for their participation and constructive feedback in the production of the thesis.

I am thankful to all other staffs at IIT, University of Dhaka for creating a pleasant work environment.

I am also thankful to the DSSE Student Research group, Institute of Information Technology, University of Dhaka, for their continuous support and valuable feedback that helped me in improving my work.

I would also like to thank the ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh, for giving me research fellowship and supporting my research work. I am also grateful to Microsoft Research for sponsoring me in <Programming> Student Research Competition, 2017.

List of Publications

1. "A Similarity-Based Method Retrieval Technique to Improve Effectiveness in Code Search" in *Proceedings of the ACM International Conference on the Art, Science, and Engineering of Programming, <Programming>,* Brussels, Belgium, 2017. [Accepted in <Programming> ACM Student Research Competition]

2. "Improving Recall in Code Search by Indexing Similar Codes under Proper Terms," in *Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC),* pp. 35-42, Hamilton, New Zealand, December 6, 2016.

3. "A Search Log Mining Based Query Expansion Technique to Improve Effectiveness in Code Search," in *Proceedings of the 19th IEEE International Conference on Computer and Information Technology (ICCIT),* Dhaka, Bangladesh, 2016.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The effectiveness of a Code Search Engine (CSE) is defined as the number of retrieved code fragments that are relevant to the user needs. The reason is that developers need existing code snippets to understand the implementation of a particular feature, to know the usage of an Application Programming Interface (API), or to reuse these with some adaptations in the development context [5]. A CSE should retrieve as many relevant code fragments as possible so that developers can choose the one that best satisfies their needs. Usually, traditional Information Retrieval (IR) centric approaches are used by the CSE. In these approaches, keywords found in the code snippets (method name, variable name, etc.) are used to construct the index [6]. In these approaches, a collection of code fragments that perform the same feature but do not contain the same keywords, will be indexed against different terms. When a query term matches one of these index terms, only the corresponding code fragment will be retrieved instead of all these fragments. Other relevant code fragments cannot be fetched due to this keyword matching policy and thus the effectiveness of the code search engine is reduced.

The chapter presents the motivation for improving the effectiveness in code search. Besides, the research question which is dealt through out this work and contribution of the research are also described. The chapter then provides an overview on the organization of the thesis for providing a guideline to the readers.

1.1 Motivational Example

Code search is performed by the developers to obtain reusable source codes from a code base. Code search engines should provide all the relevant codes against a given user query. If the engines miss relevant codes, developers have to write the codes from the scratch. This induces additional development time and cost. Let us consider the sample code fragments shown in Listing 1.1.

```

void bubbleSort(int* A, int sz) {
  for(int i=0;i<sz;i++){
    for(int j=0;j<sz-1;j++){
      if(A[j]>A[j+1]){
        int temp=A[j];
        A[j]=A[j+1];
        A[j+1]=temp;
      }
    }
  }
}

void selectionSort(int A[], int n){
  int pos_min,temp;

  for (int i=0; i < n-1; i++)
  {
    pos_min = i;

```

```
      for (int j=i+1; j < n; j++)
      {
        if (A[j] < A[pos_min])
        pos_min=j;
      }

      if (pos_min != i)
      {
        temp = A[i];
        A[i] = A[pos_min];
        A[pos_min] = temp;
      }
    }
  }

  void srt(int* A,int p,int r) {
    if(r>p) {
      int q;
      q=(p+r)/2;
      srt(A,p,q);
      srt(A,q+1,r);
      int n1=q-p+1;
      int n2=r-q;
      int L[n1+1];
      int R[n2+1];
      for(int i=0; i<n1; i++) L[i]=A[p+i];
      for(int j=0; j<n2; j++) R[j]=A[q+1+j];
      int i=0;
      int j=0;
      int n=0;
      while(i!=n1 && j!=n2) {
        if(L[i]>R[j]) {
          A[p+n]=R[j];
          j++;
```

```
        } else {
          A[p+n]=L[i];
          i++;
        }
        n++;
      }
      while(j!=n2) {
        A[p+n]=R[j];
        j++;
        n++;
      }
      while(i!=n1) {
        A[p+n]=L[i];
        i++;
        n++;
      }
    }
  }
```

Listing 1.1: Implementation of Bubble Sort, Selection Sort and Merge Sort

There are three methods in the sample code snippet. The first method named *bubbleSort* sorts a given array of integers in ascending order. The second method named *selectionSort* also performs the same task. The complexities of these methods are the same that is $O(n^2)$ where n is the size of the input array. Another method named *srt* is shown in the example code snippet. The method implements the merge sort algorithm and its complexity is $O(n \log n)$. This method is faster than the other two methods - *bubbleSort* and *selectionSort*. Since traditional code search engines follows IR-based approaches, keywords found in the method name are used as index terms. That is, *bubbleSort* will be indexed against the terms "bubble" and "sort". *selectionSort* will be indexed against the terms "selection" and "sort". Since *srt* contains a single term "srt" in the name, the term is used

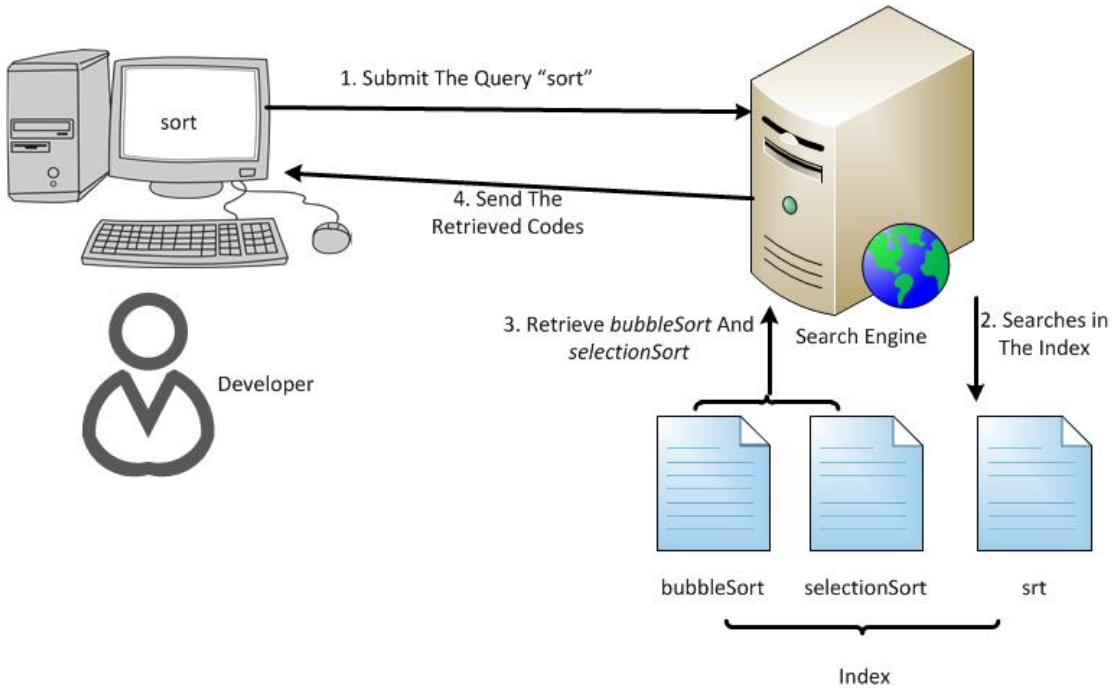

Figure 1.1: Traditional Code Search Scheme

to index the method. Assume that a developer looks for an efficient method for sorting an array. When "sort" is provided as the search query, two methods *bubbleSort* and *selectionSort* will be retrieved. The reason is that both methods are indexed against the term "sort" as shown in Figure 1.1. However, the method "srt" will not be retrieved in this case because "sort" is not used as index term for the method. The developer is deprived of the method *srt* which is more efficient than two other methods. In that case, either any of the retrieved methods - *bubbleSort* and *selectionSort* can be reused or the merge sort algorithm is needed to be implemented from the scratch. This will induce additional time and cost. The effectiveness of the code search engines should be increased so that the method "srt" is retrieved although the term is not consistent with the implemented feature. It will assist to make software development faster through finding reusable code fragments.

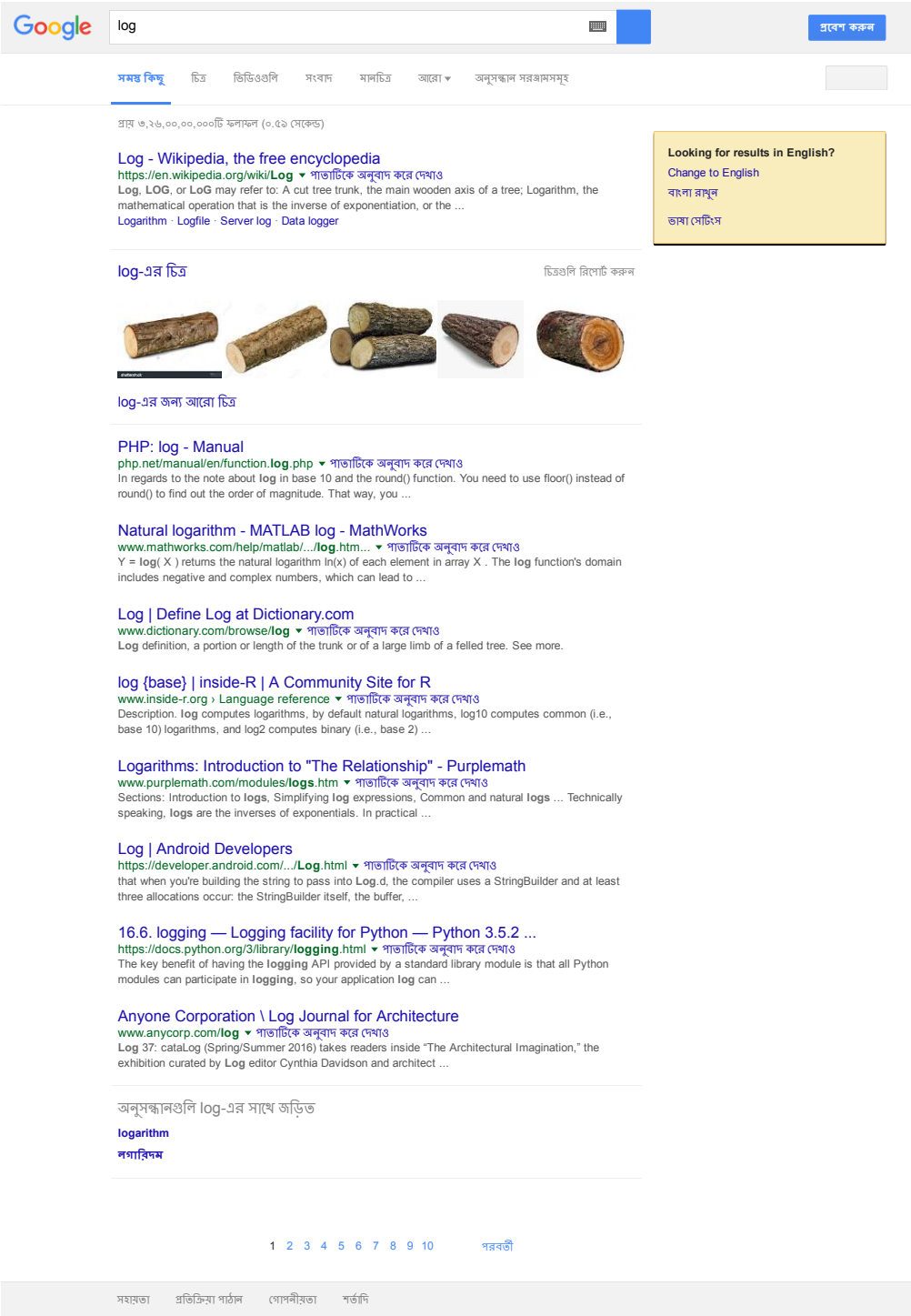

Figure 1.2: Snapshot of Google Search Results for Ambiguous Query Term

Consider another case where the search engines fail to retrieve relevant code fragments. User query comprises a set of terms to represent the search intent. Code search engines retrieve relevant codes or software artifacts based on the query terms. If the query terms are ambiguous or irrelevant to the search intent, the engines may provide irrelevant search results. An example of ambiguous query is shown in Figure 1.2. The figure depicts the search results against the query "log". There are three different semantic meanings of "log". It can be used to retrieve code snippets that perform logging of program execution. Mathematical logarithmic operations can be defined by the term. Another meaning of "log" is the trunk of a tree. When "log" is used as the query, it is difficult for the search engine to determine the search intent. In the figure, the top result is shown about the trunk of a tree against the query "log". However, the developers usually use the query to find code snippets or documentations to know how to log a program execution information. Only three results are found in the figure that are relevant to the intent. These are "PHP: Log - Manual", "Log | Android Developers", and "16.6 Logging - Logging facility for Python". All other search results shown in the figure are irrelevant to the needs of the developers. Thus, the presence of ambiguous term in the user query reduces the effectiveness of the search engines.

1.2 Issues in State-of-the-Art Approaches

Researchers have proposed various techniques to improve the effectiveness of code search engines. These techniques can be broadly classified into four types like Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Test Driven Code Search (TDCS), and Semantic Based Code Search (SBCS). Initially, KBCS [6, 5, 7, 8, 9, 10, 11, 12, 13, 14] was introduced where source code is considered as plaintext document [2]. Source codes are indexed based on the terms generated from the code and searching is performed on the index. As

this approach does not consider similarity between source code having different keywords, it cannot retrieve more relevant codes.

In order to satisfy the needs to find and reuse existing software components, IDCS [15, 16] and SBCS [17, 18] was proposed. Using these approaches, a user can define the required component interface as query, and relevant components are retrieved based on that. Usually, for searching relevant codes, IDCS uses method name, parameters, class name, etc. However, two or more code fragments may have different interfaces but perform the same task. IDCS considers that these code fragments are different due to having different interfaces. Thus, the recall of the approach is reduced.

Reusing existing components often takes a significant amount of time to understand and adapt in the development context. This is because, components are developed for reusing under a particular development context which may not suit in another context or may require modifications. To automatically find and adapt reusable components, TDCS [15, 16] and SBCS [17, 18] were proposed. TDCS is an effective technique in terms of precision which employs test cases on the retrieved codes. In this approach, most of the test cases fail not only for functional requirements mismatch but also for syntactic mismatch of the interface definition [15]. For this reason, semantically relevant code fragments could not be retrieved and hence, the performance is degraded in terms of recall. A SBCS technique was proposed which employs test cases to obtain semantic information. For a given user query, it searches in the popular search engines like google code[1], krugle[2], etc. and provides the semantically matched code fragments. SBCS depends on the search results of the code search engines and these engines use KBCS that has low effectiveness like all the approaches mentioned above.

[1]http://www.code.google.com
[2]http://www.krugle.com/

1.3 Research Questions

In order to improve the effectiveness of a code search engine, feature-wise similar code fragments should be indexed under common and proper terms. This will retrieve all the relevant code snippets together that contain different keywords. However, it is challenging to determine feature-wise similarity among the syntactically different code fragments because it requires determining the implemented features of these fragments [19]. Another challenge is to select proper term that represents the intent of a code fragment properly. For example, three methods named "bubble", "bbl" and "bubbleSort" sort an array of elements. The term "bubble" is ambiguous, "bbl" is not consistent with human perceivable language dictionary. However, the term "bubble sort" expresses the implemented feature properly and it should be used to index all these methods. Selecting such proper term automatically for indexing is also a research challenge. All these challenges lead to the following research question.

- How to increase the effectiveness in code search by indexing similar codes under proper terms?

Two sub-questions are associated with this question, which are as follows.

1. How to detect similar codes that implement the same feature?

   To answer this sub-question, following steps may be adopted:

   (a) A source code repository containing different projects needs to be built on which searching will be performed

   (b) Projects and methods of each project are required to be extracted to provide method level searching facility

   (c) An algorithm needs to be developed to identify similar methods by matching the signature and body of each method with other methods

(d) It is required to construct method clusters based on the similarity among methods to retrieve more relevant code fragments

2. How to select appropriate terms for similar codes during indexing?

   The steps outlined below may be followed to deal with this sub-question.

   (a) Keywords from the body and signature of each method in a cluster, are required to be gathered to identify candidate terms

   (b) Term frequency (tf) amd inverse document frequency (idf) need to be calculated for each keyword to select proper terms. Here, document frequency is the number of methods that contain the keyword, inverse document frequency is the logarithmic ratio of document frequency and total number of methods, and term frequency is the number of occurrences of the keyword in whole method collection

   (c) The top scored keyword(s) may be selected as term(s) to represent the similar codes

1.4 Contribution of This Research

In this thesis, a technique named Similarity Based Method Finder (SBMF) is proposed to retrieve more relevant methods from code base. An overview of the technique is shown in Figure 1.3. According to the figure, the technique first parses all the methods from the source code to construct a repository of methods. It generates data dependency graph for each method and converts the method into self-executable method that is program slice through resolving data dependency, and redefining parameters and return type. Later, all the methods are clustered into a number of clusters where methods in the same cluster perform the same task. To detect feature-wise similarity among a set of methods, signatures (that is, parameters and return types) of these methods are checked. Methods having the

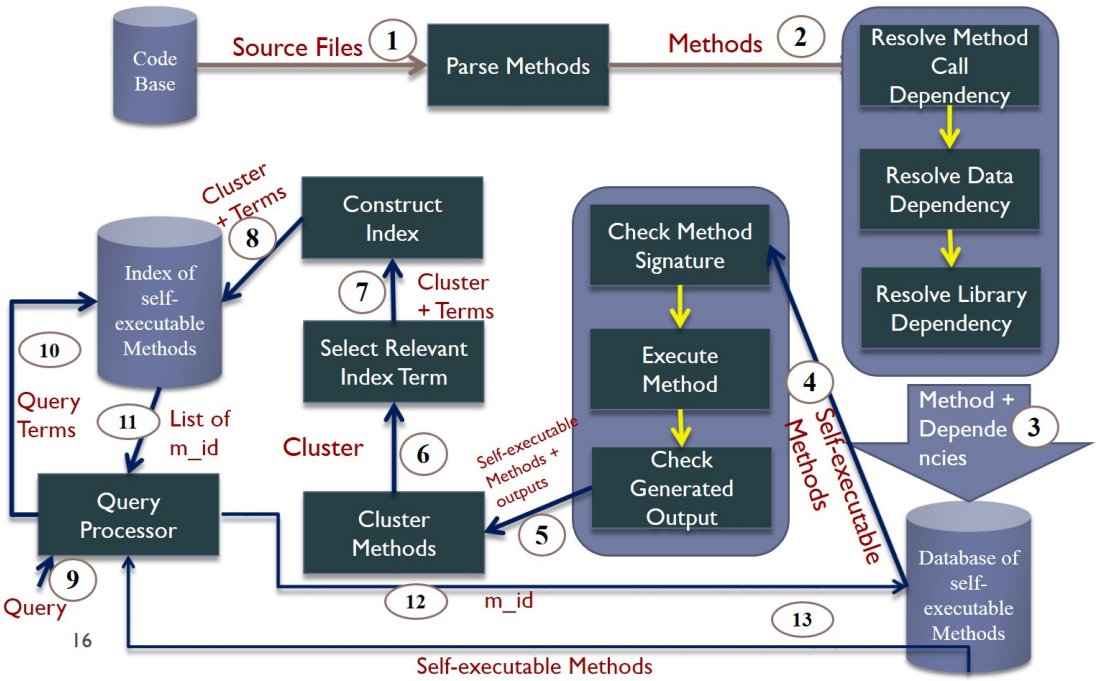

Figure 1.3: Overview of SBMF

same signature are then executed against a set of generated input values. Among these methods, those which produce the same output are considered as feature-wise similar, and a cluster is constructed to store these methods. To identify proper terms for a cluster, keywords are obtained from the methods in the cluster and method frequency is calculated for each term. Such terms are considered as representative terms if these are found in most of the methods of the cluster. All the methods of the cluster are then indexed against the terms so that these are retrieved all together if a query term matches one of these methods. At last, user query is expanded by adding topic and context specific terms to increase the matching probability between the query terms and index terms. The technique mines the search engine usage logs to select the terms for the expansion.

In order to evaluate the proposed technique, a tool was developed. Two types of code search techniques, KBCS and IDCS, were compared with SBMF to show its efficiency. An existing system named Sourcerer [11] was used for the implementation of KBCS and IDCS. However, SBCS and TDCS were not considered for

comparison, because these techniques use KBCS to construct the index of source code. In these techniques, initial search results are obtained by using the keyword matching scheme. Next, the results are filtered based on the semantics of the development context. For comparative result analysis, five metrics were used which are *recall*, *number of methods retrieved* and *feature successfulness, self-executability,* and *precision@10*. Here, *feature successfulness* determines whether at least one relevant method is retrieved against user queries provided for a feature. In the context of this paper, *a feature can be considered as a requirement given to a developer to implement*. 50 open source projects were selected to carry out the experiment. The result analysis shows that on an average SBMF increases recall by 38% and 58% more than KBCS and IDCS, respectively against 195 queries. Besides, SBMF is successful for all the features whereas KBCS and IDCS are successful for 7 and 3 features out of 9 respectively. All the methods retrieved by SBMF are self-executable. The query expansion process of SBMF shows 48.6% more effectiveness in terms of precision at 10 (precision@10) than another existing technique named Thesaurus-based Automatic Query Expansion [20].

1.5 Organization of the Thesis

This section provides an overview about the remaining chapters of this thesis. The chapters are organized as follows.

Chapter 2: This chapter discusses different types of code search and the necessity of code search in software development. The basic architecture of the code search engines is constructed following Information Retrieval (IR) based approaches. So, a notion on IR is provided in the chapter. Different code search engines and the usage of each engine are also presented. The architecture and data model of common code search engines are explained at the end of the chapter.

Chapter 3: This chapter provides a brief literature review on code search.

Existing techniques to improve the effectiveness in code search are discussed. The pros and cons of the techniques are also presented in the chapter.

Chapter 4: In order to improve the effectiveness in code search, a technique named SBMF is proposed in this thesis. This chapter presents the detailed description of the technique. All the devised algorithms for SBMF are also explained in the chapter.

Chapter 5: This chapter provides the mathematical analysis of the proposed technique. The chapter first presents the mathematical models of the problems that decrease the effectiveness of code search engines. For each of the problems, the resolution mechanism of SBMF is shown mathematically in the chapter.

Chapter 6: A detailed explanation on the implementation of the proposed technique is presented in this chapter. The experimental setup and dataset used to evaluate the technique are provided in the chapter. A comparative result analysis is shown to justify the effectiveness of the technique in retrieving relevant codes.

Chapter 7: It is the concluding chapter of the thesis. The chapter contains a discussion about the proposed technique, important threats to validity and some future directions.

Chapter 2

Background Study

Code search is a commonly performed activity by the developers [21, 22]. Recent studies showed that developers spent 19% of total development time in code search [23]. Since a huge amount of open source code is available in the internet [24, 25], developers like to reuse these codes instead of writing from the beginning [26]. Usually, developers perform code search to understand the usage of an Application Programming Interface (API), implement a feature, reuse code snippets and so on [27]. Code search engines are developed to satisfy the user needs by retrieving relevant code fragments. In this chapter, the motivations behind code search are discussed to understand the importance of the activity in software development. For different search intents, developers look for code fragments in various granular levels such as method, class, components, etc [28]. So, classification of code search and the respective search intents are discussed in this chapter. The basic architectures of code search engines are built based on Information Retrieval (IR) based approaches [8, 29]. This chapter provides a notion on IR to understand the indexing and query formulation mechanisms in the engines. To show the necessity of code search engines, this chapter presents the differences between general purpose search and code search. At the end of the chapter, a relationship between code search and software reuse is drawn.

2.1 Code Search

Code search involves finding source codes in order to assist software development and maintenance tasks, in particular code reuse [24, 30, 31]. It is considered as a critical part of software development [32]. A study on software engineering work practices showed that searching was one of the most common activities for software engineers [21]. Developers may perform code search on a local project or a large code base. In a local project, developers search specific code fragments to locate bugs [33, 34, 35], find code smells [36], detect code clones [37, 38], etc. Developers look for sample codes in a large code base when they need to reuse existing codes or to know how to implement a feature [26]. Current software development trend is to develop a software through reusing existing codes as much as possible [39, 40, 41]. Code search assists in this regard by finding relevant source codes.

2.1.1 Types of Code Search

Code search plays a crucial role in code reuse. The reason is that developers first look for existing codes while implementing a feature. If they do not find any relevant codes, they write code from the scratch [39]. In terms of code reuse, code search can be classified into two types - component search and code snippet search. These types are explained as follows.

Component Search

A software component is a software module or package that encapsulates a set of functionalities and data [42]. A software system consists of different software components. Usually, the scope of a software component varies from one context to another [43]. For example, according to the view point of a developer, a component may be a library, a set of classes, etc. From the perspective of an UX designer, a component of the system may be a form, navigation, etc. When the developers

start developing a component, they usually search for the similar components. They investigate the functionalities of the components to directly reuse the best one. Apart from the functionalities, they also check the applicability and license issues related to the components [44]. In order to plug in the development context, the suitable component is further customized by the developers. This type of reuse helps to reduce software development time and effort. Different code search engines [5, 11, 45, 46, 47, 48] have been proposed in the literature to find relevant component from a large code base.

Snippet Search

Snippet search involves retrieving code fragments to know the usage of an API. Other than that, developers look for code snippets to reuse with little or no modifications. Snippet search becomes a common activity when developers need example codes to remember or learn the syntax of a language. A snippet usually comprises few lines of code that performs a specific task. For example, a developer may look for a code snippet to know the implementation of binary search algorithm. A relevant code snippet of binary search is shown in Listing 2.1.

```
int binarySearch(int arr[], int l, int r, int x)
{
  if (r >= l)
  {
    int mid = l + (r - l)/2;
    if (arr[mid] == x)
       return mid;
    if (arr[mid] > x)
      return binarySearch(arr, l, mid-1, x);
    return binarySearch(arr, mid+1, r, x);
  }
  return -1;}
```

Listing 2.1: Code Snippet of Binary Search

2.1.2 Motivations Behind Code Search

Since the end of 1990, one of the most common practices of software engineers has been to search for source code [21]. Developers look for existing source code for component reuse and snippet remixing. Component reuse involves using existing suitable components with some adaptations to implement a system. Snippet remixing is concerned with the use of small code fragments by copying and pasting in the development context. Now-a-days it is said that everything is a remix[1]. A significant amount of code that is written today has already been written by someone before. The reason is that huge amount of open source code are being exposed everyday in the web by the open source developers. Current development practice encourages software developers to reuse existing codes as much as possible. Developers are only encouraged to write from scratch if the codes are not available in the web. For this reason, a large part of software is developed by using code from the open source software. Programming is now viewed as remixing codes of different software systems.

While performing code search, three types of motivations are seen among the developers [26]. These are discussed as follows.

Understanding the Usage of an API

API is one of the most common forms of software reuse [49]. Usually, developers are more interested to reuse existing libraries or components rather than the implementation of the libraries. API comprises functionalities that are exposed to the developers to reuse these for developing or accomplishing a particular task [50]. Often a function defined in an API needs to invoke another function to perform a certain task. Such sequence of function invocation is known as API usage pattern [51]. In order to gain insight about different usage patterns, developers look for documentation, example codes and other resources. Documentation is considered

[1] http://www.everythingisaremix.info/

as one of the most important sources of information to understand API usage patterns. However, in most of the cases, API documents do not contain detailed and up-to-date information [52]. Even, most of the APIs do not have concise documentation due to poor project management or other reasons. In addition, these documents are not updated with the passage of time and changes of the API libraries. So, developers find very little information about API usage patterns which contributes a little to the developers for reusing API functionalities. The next source of information is collection of example codes in which API functions are used for the implementation of different tasks or features. Developers search for the example codes to know how to use the functionalities exposed by the API [53].

Understanding the Implementation of a Feature

A common practice among the developers is that they search for source codes to know the implementation of a feature. At the beginning phase of software industry, developers had to write code from the scratch. Later, they tested the codes to ensure the correctness of the program. Due to increasing popularity of open source software development, developers in the open source community have built a rich code base and day-by-day the size of the code base is increasing [54]. More quality codes are being added to the code base continuously. Millions of features have been implemented in different ways by various developers [55]. While developing a software system, developers need to implement various requirements given by the clients. A developer may not have prior knowledge about the implementation of the feature. In this case, existing codes that implement the feature or related to it, are considered as good source of information.

Finding Source Code for Direct Reuse

In order to reduce software development time and effort, developers search relevant software components in the open source software systems. They select the suitable components and plug these in the development context. Another type of reuse is that developers directly copy and paste code snippets to implement the functionalities assigned to them. While working on a requirement, developers first look into their local repositories to find relevant codes. If no such code fragment is found, they search in the open source repositories. The tendency of the developers towards code reuse clearly states the importance of code search in faster and cost-effective software development.

2.2 Information Retrieval

Information retrieval (IR) is defined as finding material of an unstructured nature (usually text) that satisfies an information-need from within large collections [56]. Billions of data are available on the internet which contain a lot of information [57]. IR is concerned with finding relevant information from this vast ocean of data and satisfying information-needs of the users. The process of IR involves following three stages.

- Collection of Data Representation
- User Query
- Retrieval of Relevant Information

Key concepts of IR along with basic indexing and query processing techniques are discussed in the following subsections.

Table 2.1: Sample Documents

DocId	Terms
D1	sort, array, ascending, order
D2	sort, implementation, bubble
D3	sort, array, bubble

2.2.1 Term, Document and Posting List

In IR, a *term* is defined as a keyword or group of words that have some meaning. For example, each word in an article can be considered as a term. A *document* in IR is a specific unit of retrieval such as a web page, an article, a book, a section in a book, etc. In other words, a document can be seen as a collection of terms. In order to index the document, a *term-document incidence* matrix is constructed. Matrix element (t, d) is '1' if document in column d contains the term in row t. Otherwise, the value will be '0'. In order to understand how the matrix is constructed, a sample term-document incidence matrix for Table 2.1 is shown in Table 2.2. In Table 2.1, there are three documents denoted by D1, D2 and D3. D1 has four terms - "sort", "array", "ascending" and "order". Again, document D2 has three terms - "sort", "implementation" and "bubble". Document D3 has also three terms - "sort", "array" and "bubble". It is seen that the term "sort" is common in all three documents. So, the row for "sort" in the term-document incident matrix contains '1' for all the three documents. Similarly, the term "implementation" only appears in D3, so the cell (implementation, D2) is set to '1'. As "implementation" does not exist in D1, the cell (implementation, D1) is set to '0'. This is how the term-document incidence matrix is constructed.

An *inverted index* is the opposite of term-document incidence matrix. It maps each term to a set of documents. In IR, the set of documents for a given term is called as *posting list* for the term and each document is called a *posting* in the inverted index. The collection of all terms in the inverted index is called *dictionary*. An inverted index is shown in Figure 2.1 for the document collection in Table

Table 2.2: Term-Document Incidence Matrix for Table 2.1

	D1	D2	D3
sort	1	1	1
array	1	0	1
ascending	1	0	0
order	1	0	0
implementation	0	1	0
bubble	0	1	1

2.1. The dictionary of the inverted index comprises the terms - “sort”, “array”, “ascending”, “order”, implementation” and “bubble.” For the term “sort”, the respective posting list contains D1, D2 and D3 because the term exists in all the three documents.

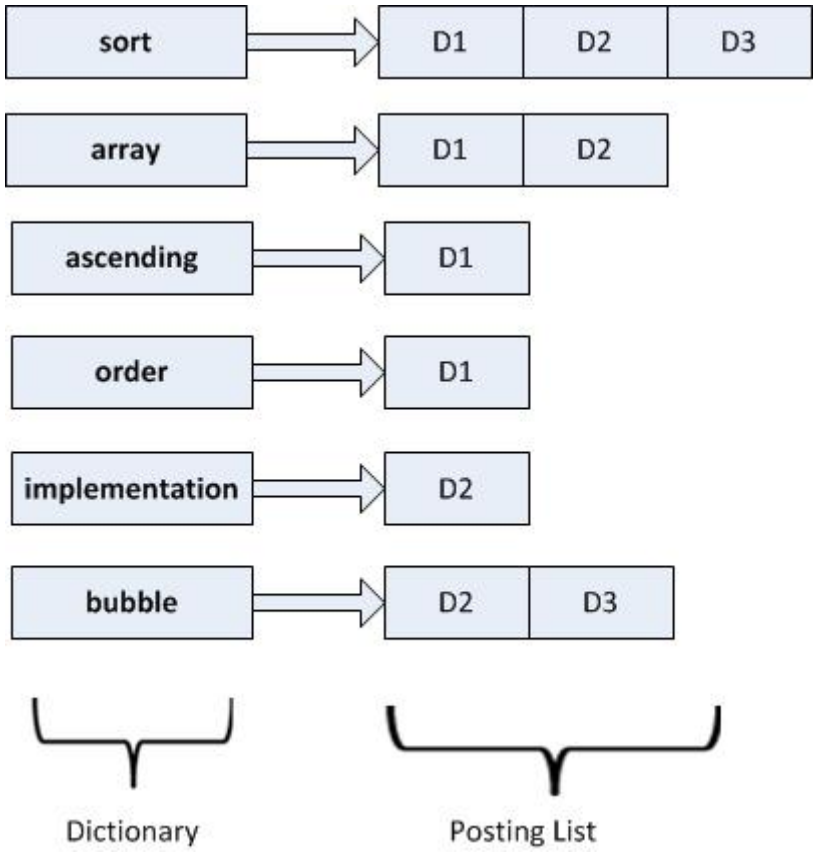

Figure 2.1: Inverted Index for Table 2.2

2.2.2 Linguistic Preprocessing

In order to construct the vocabulary of terms, IR systems deal with some basic linguistic issues of tokenization and linguistic preprocessing. A document can be considered as a stream of characters as said earlier. Tokenization is a process that chops the character streams into tokens. Later, linguistic preprocessing is performed on the tokens to generate index terms.

Tokenization

Before indexing a collection of documents, the unit of document is selected to generate terms. For a given unit and a character sequence that is document, tokenization chops it into pieces which are termed as tokens. A token can be defined as an instance of character stream in a document. A set of semantically relevant tokens is called a *type* or *class* in IR. These types are normalized to generate terms.

Usually tokens are generated by removing whitespace and punctuations from a given character sequence. An example is shown in Figure 2.2.

Input Text: Bubble sort, selection sort, merge sort are all the sorting algorithms

Tokens:	bubble	sort	selection	sort	merge	sort	are	all	the	sorting	algorithms

Figure 2.2: Tokenization

While indexing source code files, textual words such as method name, variable name, class name, etc. are tokenized by code search engines. These names are written following different naming conventions such as camel case, pascal case, etc. These naming conventions are also used to generate tokens. For instance, the naming convention of a method name in camel case is that first letter of the name will be small letter. If the name comprises more than one word, the first letter of each word will be capital except the first word of the name. Following

Table 2.3: Sample Stop Words in Reuters-RCV1 [1]

a	an	and	are	as	at	be	by	for	from
has	he	in	is	it	its	of	on	that	the
to	was	were	will	with					

this convention if a method name is *addNumbers*, the tokens will be "add" and "numbers".

Stop Words Removal

All the words in a document are not equally important. Some words are found that contribute little to express the topic. Most of the cases, these words do not have any semantic meaning such as prepositions in a sentence. Since these words commonly occur in many documents, semantically different documents cannot be distinguished by these words. In IR these words are termed as *stop words*. A list of 25 semantically non-selective stop words in Reuters-RCV1 [1] is shown in Table 2.3.

Web search engines do not consider stop words in a document. The reason is that adding stop words to the dictionary increases the size of the index unnecessarily. Stop words should be identified and removed from the index because the absence of these words induces less harm than the existence in the index. Removing stop words from the index significantly reduces the size of the posting list. The general approach to determine the stop word list is to find most frequently occurred terms in the index. It can be done by sorting the term list based on *collection frequency* that is number of times a term occurs in the document collection. Later, the identified words are discarded while constructing the index of documents.

Normalization

Information retrieval systems match query terms with the index terms to retrieve relevant documents. However, there are many cases when two tokens or character sequences are not exactly the same but both denote the same semantics. For instance, *preprocess* and *pre-process* are semantically the same but the character sequences are different. The reason is that one of the tokens contains a hyphen. In order to retrieve the documents indexed by *preprocess* or *pre-process*, both tokens need to be converted into a common type or class. Normalization helps in this regard. *Token Normalization* is a process that canonicalizes tokens having minor differences in the character sequences of the tokens. IR systems employ normalization to construct equivalent classes of the tokens. Each class consists of unnormalized tokens. The name of class is determined by the normalized form of the member token. A set of rules are also defined to perform normalization. For example, removing hyphens from the tokens is one of the commonly used rules in IR. If the hyphen is removed from the token *pre-process*, both tokens will be the same. The equivalent class for *pre-process* and *preprocess* will be *preprocess* and it will be then used as index term. When a query term will match this token, documents containing either *preprocess* or *pre-process* will be retrieved.

Another common strategy of normalization is *case-folding*. In this approach, all letters in a token are converted to lower case letters. For instance, *Automatic* and *automatic* are two semantically the same tokens. However, these are different in terms of case sensitivity. The reason is that *Automatic* starts with an upper case letter 'A' and *automatic* starts with a lower case letter 'a'. Here 'a' and 'A' are different due to the case sensitivity. In this scenario, case-folding converts both tokens into lower case format that is *automatic*. As a result, documents having *automatic* or *Automatic* will be retrieved simultaneously.

Stemming

Due to grammatical issues, different forms of a word are seen in different documents, for example, *go, goes, going, went, gone,* etc. All these forms of the word are semantically similar but these are used to represent various contexts. It is useful to have all the documents when any of the words matches to the query term. That is, if a query term matches to *go*, IR systems should retrieve documents that contain any of the words. Stemming assists to achieve this goal.

In IR, stemming is the process that generates the root form of a given word. The aim of the process is to reduce inflectional forms and derivative forms of a word to a common base form. For instance,

$$\text{am, are, is} \implies \textit{be}$$

$$\text{car, cars, car's, cars'} \implies \textit{car}$$

$$\text{go, went, gone, goes} \implies \textit{go}$$

Using these mappings, a sample text, *he is going to hire a car* can be converted as follows.

$$\text{he is going to hire a car} \implies \text{he be go to hire a car}$$

There are many open source and commercial stemmers available such as *Porter stemmer [58, 59], Lovins stemmer [60], paice stemmer [61],* etc. Among these stemmers, *Porter stemmer* is most commonly used in IR systems. Studies showed that it is empirically very effective. The technique uses a set of rules to stem tokens in a document. These rules are exemplified in Table 2.4.

Table 2.4: Rules of Porter Stemmer with Examples

Rule			**Example**		
SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
SS	→	SS	caress	→	caress
S	→		cats	→	cats

2.2.3 Index Construction

IR systems build index in advance to gain the speed benefits in retrieval. To construct the index for a given collection of documents, the basic steps are given below.

(a) Collect the documents for which index will be constructed

(b) Tokenize the text or stream of characters in each of the documents

(c) Convert each document into a list of tokens

(d) Remove stop words from the list of tokens to reduce index size

(e) Perform normalization on the tokens to obtain a list of normalized tokens

(f) Stem the normalized tokens to convert each token into its root or base form

(g) Index the documents by generating inverted index that is posting list and dictionary

2.2.4 Scoring Schemes in Information Retrieval

As the number of documents is huge and it is increasing rapidly day-by-day, huge number of relevant documents are retrieved against a search query. It is not possible at all for the users to read all the relevant documents at a time. Users want to get the most relevant document at the top of the search results. So, IR systems need to rank the search results according to the relevance. Several scoring

schemes are used in IR. Some basic and commonly used schemes are explained as follows.

Term Frequency

Term frequency is the number of occurrences of a term t in a document d. It is defined as $tf_{t,d}$. Consider the following document.

*three algorithms are bubble **sort**, merge **sort**, selection **sort***

In the document, the term *sort* occurs three time. So term frequency of *sort* is three.

Inverse Document Frequency

Term frequency as discussed above considers all the terms equally important. However, in many cases, some words are found that cannot differentiate semantically different documents at all. For this reason, a new scheme named *inverse document frequency* is used in IR systems. Inverse document frequency *idf* for a term is calculated using *collection frequency* and *document frequency*. *Collection frequency* is defined as the number of occurrences of a term in the collection of documents. *Document frequency* of a term is the number of documents in the collection that contain the term. In order to understand the metrics, consider the following document collection which comprises three documents - D1, D2 and D3.

*D1: binary **search** tree algorithm*

*D2: binary **search** and linear **search***

*D3: ternary **search** algorithm*

As the term *search* appears four times in the document collection, so collection

frequency of the term is four. Again, all the three documents contain the term, so document frequency of *search* is three.

If the document frequency of a term t is denoted by df_t and total number of document in the collection is N, inverse document frequency idf_t can be defined as follows.

$$idf_t = \log \frac{N}{df_t} \tag{2.1}$$

Tf-idf Weighting Scheme

Sometimes, IR systems combine term frequency and inverse document frequency to determine the weight of a term. This composite weighting scheme is defined by the following equation.

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t \tag{2.2}$$

Here, $tf\text{-}idf_{t,d}$ assigns a weight to a term t in document d and the weight is

(a) highest when t appears frequently in a small number of documents

(b) lowest when t appears in almost all the documents

(c) lower when t occurs in many documents or appears many times in a single document

$tf\text{-}idf$ is used to calculate score for a document against a user query. For a given query q and a document d, the score calculation formula is given below.

$$Score(q, d) = \sum_{t \in q} tf\text{-}idf_{t,d} \tag{2.3}$$

Vector Space Model for Scoring

One of the most common schemes to calculate similarity score between two documents is *vector space model*. It is also used to compute similarity score between a user query and a document. In this scheme, a document is represented as a vector where dimensions are denoted by the terms in the document. Value in each dimension is calculated using tf-idf scoring scheme. The similarity score between two documents d_1 and d_2, is computed using *cosine similarity* between two documents. If $\vec{V}(d1)$ and $\vec{V}(d2)$ are vector representation of d_1 and d_1 respectively, the cosine similarity score is calculated using the following equation.

$$sim(d_1, d_2) = \frac{\vec{V}(d1) \cdot \vec{V}(d2)}{|\vec{V}(d1)||\vec{V}(d2)|} \tag{2.4}$$

Here, the numerator represents the dot product of the vectors $\vec{V}(d1)$ and $\vec{V}(d2)$, and the denominator is the product of their Euclidean lengths. For two vectors $\vec{x}$ and $\vec{y}$, the dot product $\vec{x} \cdot \vec{y}$ is defined as $\sum_{i=1}^{M} x_i y_i$. If $\vec{V}(d)$ is a document vector for a given document d with M components $\vec{V_1}(d)...\vec{V_M}(d)$, the *Euclidean* length for $\vec{V}(d)$ is defined as $\sqrt{\sum_{i=1}^{M} \vec{V_i^2}(d)}$. That is,

$$|\vec{V}(d)| = \sqrt{\vec{V_1^2}(d) + \vec{V_2^2}(d) + \vec{V_3^2}(d) + ... + \vec{V_M^2}(d)} \tag{2.5}$$

Similarly, cosine similarity score can also be calculated between a query q and a document d. Higher similarity score denotes the document is more relevant for the query. If $\vec{q}$ and $\vec{d}$ are the vector forms of q and d, the dot similarity score can be calculated using the following equation.

$$sim(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|} \tag{2.6}$$

Other tf-idf Functions

A term may occur hundred times in a document and it may appear single time in another document. It is unlikely that first document is hundred times significant than the second one with respect to the occurrences of the term. To alleviate this problem, a weighting scheme named *sublinear tf scaling* has been proposed in the literature [62]. Instead of using $tf_{t,d}$ directly, a new weight $wf_{t,d}$ is calculated using the following equation.

$$wf_{t,d} = \begin{cases} 1 + \log tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

So, the new weighting scheme can be calculated by replacing $tf_{t,d}$ with $wf_{t,d}$ in $tf\text{-}idf_{t,d}$ weighting equation. That is,

$$wf\text{-}idf_{t,d} = wf_{t,d} \times idf_t \tag{2.8}$$

Maximum tf normalization is a well-known studied technique that normalizes the *tf* weights of all the terms occurring in a document by the maximum tf in that document. Assume that, for each document d, let $tf_{max}(d) = max_{\tau \in d} tf_{\tau,d}$ where τ ranges over all the terms in d. The normalized term frequency for each term t in document d is calculated by the equation given below.

$$ntf_{t,d} = a + (1 - a)\frac{tf_{t,d}}{tf_{max}(d)'} \tag{2.9}$$

The term a in the equation is a *smoothing* term whose role is to damp the contribution of the second term - which may be viewed as a scaling down of tf by the largest tf value in d.

Table 2.5: A Sample Term-Document Incidence Matrix

	doc1	doc2	doc3	doc4
sort	1	1	1	1
bubble	1	0	1	0
merge	0	1	1	0

2.2.5 Boolean Retrieval Model

Boolean retrieval model is a basic information retrieval model where query can be submitted in an IR system in the form of a boolean expression. A user query usually comprises a set of terms. Using boolean expression, a user can employ different boolean operations like *AND, OR, NOT*. Consider a sample term-document incidence matrix as shown Table 2.5. There are four documents - doc1, doc2, doc3 and doc4 in the matrix. The documents are indexed against the terms - sort, bubble, and merge. The value of cell is set to '1' if the respective document contains the term. Otherwise, it is set to '0' according to property of *term-document incidence matrix* as stated earlier. In the table the cell *(sort, doc1)* is set to '1' as "sort" exists in "doc1". Assume the following boolean query.

sort AND bubble AND NOT merge

The query is used for retrieving documents that contain both the terms - "sort' and "bubble", but do not contain "merge". In order to retrieve relevant documents, respective row of each query term is retrieved. In Table 2.5, the first row is denoted by "sort" and the value is "1111". After that, boolean operation is performed as given in the query. For the sample query, the aforementioned steps are shown as follows.

$$1111 \text{ AND } 1010 \text{ AND NOT } 0110$$

$$= 1111 \text{ AND } 1010 \text{ AND } 1001$$

$$= 1000$$

That is, the first bit contains '1' and thus the *doc*1 will be delivered as search result against the user query.

2.2.6 Lucene

Apache Lucene is a free open-source information retrieval based software library which is written in Java [63, 64]. Lucene achieves fast search responses because it searches an index instead of searching the text directly. Here, inverted index is used, because multiple documents may be indexed against the same key. By only considering keyword instead of traversing the whole document, the desired results can be obtained.

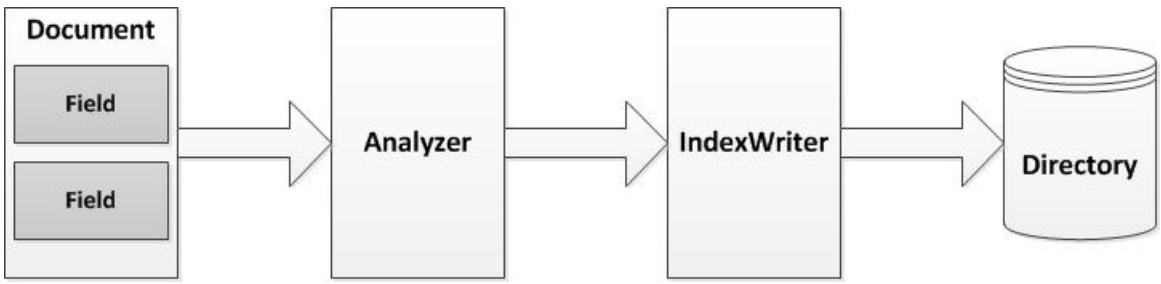

Figure 2.3: Indexing Process of Lucene

Figure 2.3 illustrates the indexing process. At first, documents are given with multiple fields. To create a document, key value pairs have been generated containing keys as names and values as contents to be indexed. The fields are analyzed by the Analyzer. After analyzing the documents, IndexWriter creates index or update the index. Among these four steps, Indexwriter plays a vital role in the process of indexing.

2.3 Code Search Engines

Code search is now considered as one of the most common activities in software development [43, 65, 66]. Developers perform code search to obtain reusable code snippets, software components, metadata of a project, etc [67, 68]. Considering these needs, different code search engines or tools are developed. This section presents the classification of code search engines to know the purposes of the engines. Besides, software artifact representation techniques are discussed to understand how the search engines store and index software artifacts. Two commonly used code retrieval infrastructures named *Krugle* and *Sourcerer* are also discussed at the end of the section.

2.3.1 Classification of Code Search Engines

As stated earlier, code search can be classified into two types which are -

(a) Component search

(b) Snippet search

Some code search engines found in the literature that can retrieve software components. Some code search engines support snippet search. Based on the retrieval units - component and snippet, a classification of existing code search engines or tools are shown in Table 2.6. The tools in the second column of the table support snippet retrieval. Tools or code search engines that support component retrieval are shown in the third column. Some tools have been found that can retrieve both software components and snippets based on the user needs. These tools are shown in fourth column of the table.

Table 2.6: Classification of Code Search Engines

	Snippet	Component	Both
API and example code search	Prospector [69] Strathcona [70] MAPO [72] Mica [73] XSnippet [18] PARSEWeb [74] STeP_IN_Java [75] Blueprint [76] SAS [77]	JSearch [6] XFinder [71]	Assieme [12]
	SNIFF [78]		
Web-based code search engine		Agora [48] SPARS-J [47] JBender [46]	Koders [79] Google [80] Krugle [3] Merobase [5] Sourcerer [11] S6 [17] Exemplar [45]
Test-driven code search		Code Conjurer [5] CodeGenie [82]	Extreme harvestin
Code snippet web search engine	Sniplr [83] Smipple [84]		
Project hosting site		SourceForge [85]	Github [86]
Reuse opportunity recommender	CodeBroker [87] Rascal [88]		
Source code integration	Jigsaw [89]	Gilligan [70]	
Others	Codetrail [90]		JIRISS [91]

2.3.2 Code Retrieval Approaches

According to the systematic survey conducted by Mili et al., six types of retrieval approaches are seen in the literature [92]. These are given below.

(a) **Information retrieval methods:** In these methods, source codes are considered as plain text document. Textual words in the source code like method name, class name, variable name, etc. are used to construct the index of codes. Keyword matching is performed between user query and index terms to retrieve relevant codes.

(b) **Descriptive methods:** These methods follow the bibliographic retrieval approaches which are commonly used in library science. These approaches use external metadata or software artifacts such as language, platform, design documents, etc. that have information about the source codes to be indexed.

(c) **Denotational semantics methods:** These approaches employ signature matching to retrieve relevant codes from a code base. Developers specify their queries by providing the signature of their required components or snippets. Usually, formal methods are used to define the signature. However, it induces additional time and effort to represent the query using formal specifications. According to the searching behavior, developers like to use free text box rather than formal specifications.

(d) **Operational semantics methods:** Here, developers provide example to represent their search intents. Usually, exemplary input values are obtained from the users, and syntactically similar code fragments are executed against the values. Although these techniques are computationally expensive, these are commonly used to retrieve workable code snippet.

(e) **Structural methods:** These methods consider the structural patterns of source codes. For a given pattern as a user query, the techniques find code snippets that follow the pattern. However, the formulation of user query in these approaches, is still a research challenge.

(f) **Topological methods:** These methods are more concerned with the ranking of retrieved source codes. To show the most relevant code fragment at the top of the search results, these methods use different metrics, for example, complexity, naming convention, structure, etc.

2.3.3 Sample Data Model of a Code Search Engine

A source code file is not so unstructured like a textual document. There exists some relationships among different parts in the source code. For instance, A project developed in Java has a collection of packages. Each package consists of some classes. Methods and fields are the elements in a class. A method can be broken down into four parts such as name, parameters, return type, and body. In order to provide precise searching facility, code search engines need to store all these information. Relational Database Modeling (RDBM) assists in this regard. In RDBM, various data are obtained from the source codes under different granular levels. As an example, the data model of a code search engine named Merobase [5] is shown in Figure 2.4.

In the figure, it is seen that the data model comprises six tables - *Project, MClass, Constructor, Dependency, Method* and *Parameter.* The table *Project* stores different information of a project such as name of the project, version, source url, etc. *MClass* stores the class name, language, package name, license, author information, comments, super class information, etc. The table Constructor is used to save the name, access modifier, parameters of a constructor. In Object Oriented Concept (OOC), a class has one or more constructors. So, there is

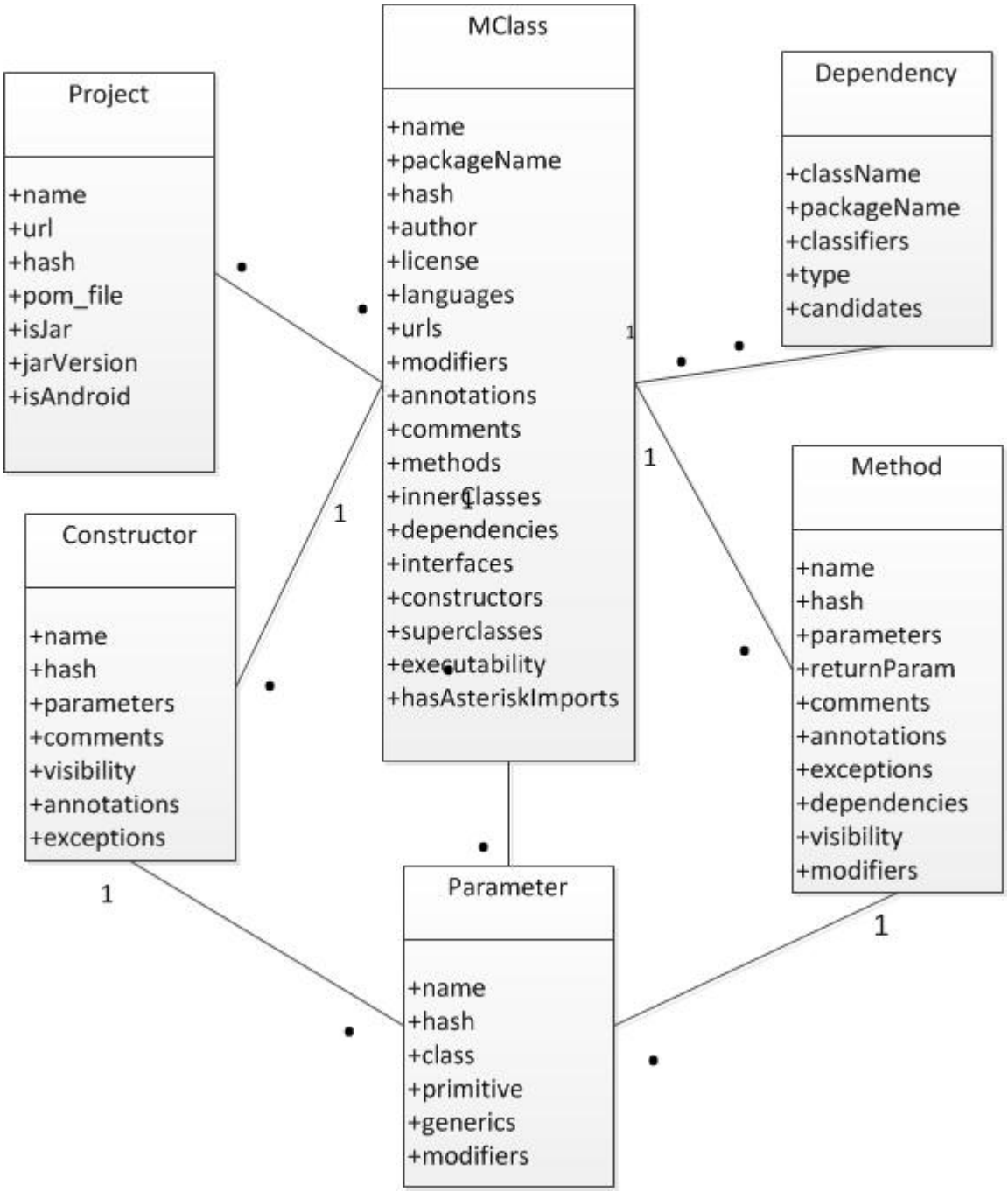

Figure 2.4: Data Model of Merobase [4]

one-to-many relationship between the tables *Constructor* and *MClass* as shown in the figure. The table *Method* is responsible to contain the name, annotations, comments, dependencies, etc. of a method. Constructor and method, both contain one or more parameters in the respective signature. To store each parameter in the database, a table named *Parameter* is seen in the model. *Parameter* is

then connected to *Constructor* and *Method* to represent one-to-many relationship. Since a class may depend on other classes, dependent class information are stored in the table *Dependency*.

2.3.4 Krugle Code Search Architecture

Krugle is one of the most commonly used code search engines that has an index of 3,500 open source projects. It also has an enterprise version to provide search facility in a private code base. There are 400 million lines of codes available in the index of Krugle. Besides, the engine has also indexed other software artifacts like project descriptions, configuration management documents, license, etc. In order to understand the basic architecture of a code search engine, Krugle is described as follows.

In Krugle, three types of information are collected which are web pages, source code and project description. The web page and source code are processed by the early version of Hadoop that runs on a cluster of fourteen slave servers and one master. The project descriptions are extracted and processed on a single server. A dedicated server is used to store these information in a MySQL database. The collection, processing and searching mechanism of each type is discussed below.

- **Web Page Crawling:** In Krugle, an open source crawler named Nutch is used to crawl technical web pages from the web. In the search engine, Nutch is customized to crawl the web pages that contain useful information for the developers. Figure 2.5 demonstrates the web page crawling process of Krugle. Initially, a collection of URLs are stored in the *URL State* database. In each iteration, unprocessed URLs are retrieved from the database and link score is calculated for each URL. Later, top scored URL is picked and the web page of the URL is fetched. Page score is calculated for the retrieved web page. Outbound links are extracted from the page and link score is assigned to each of the links. These links are then stored in the database.

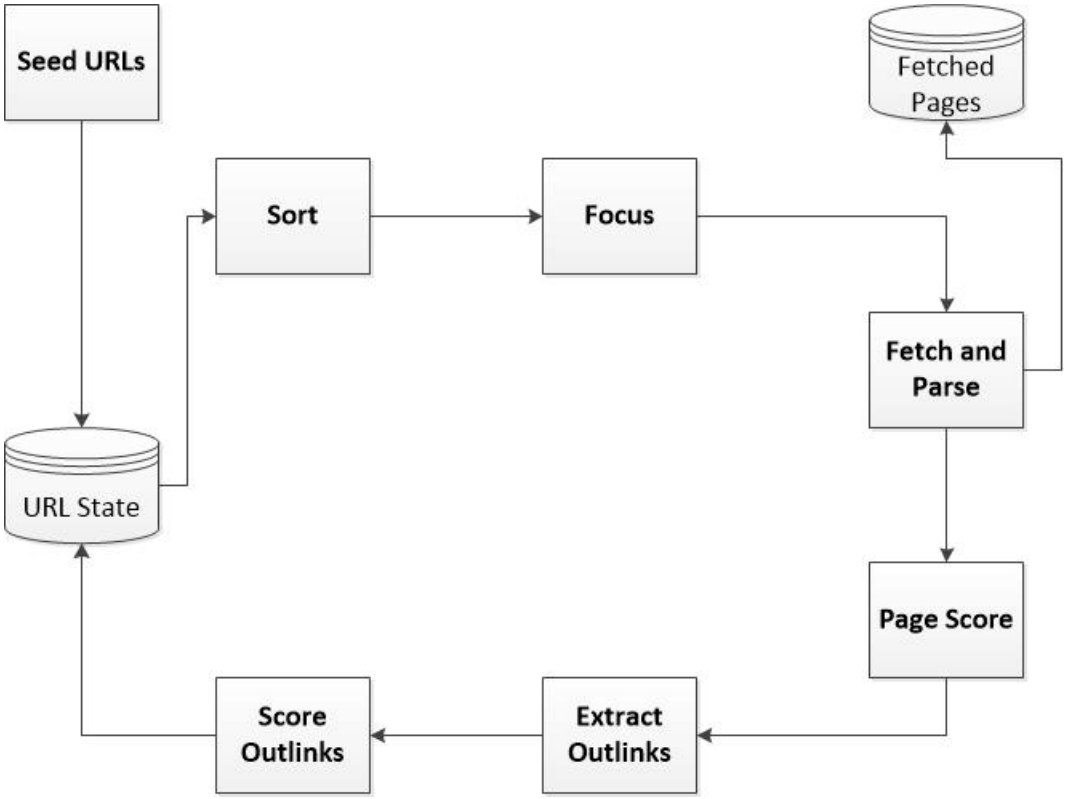

Figure 2.5: Crawling Process in Krugle

- **Web Page Processing:** Web pages fetched from the previous step, are parsed to obtain title and text. In addition, source codes in the page are also extracted from the page.

- **Web Page Searching:** Search support provided by Nutch, is employed to search the web pages. For large scale search facility, Nutch distributes the search query to multiple searchers. Each searcher searches in a slice of the index against the user query. The results of all the searchers are combined to produce the final search results.

- **Source Code Crawling:** The source code crawler is also built on Nutch. A protocol handler for Concurrent Versioning System (CVS) and Apache Subversion (SVN), is used for distributed fetching and parsing support. The crawler identifies the source code in the web page and crawls the entire trunk. Important tags are also retrieved from the web page.

- **Source Code Processing:** After obtaining source codes from the web pages, ANTLR [93] is used to construct Abstract Syntax Tree (AST). Different information like method name, class name, method signature, dependencies, etc. are obtained from the AST.

- **Source Code Searching:** Lucene is used to search source codes against a user query. When a user query is submitted to Krugle, the query is distributed among the code searchers that are built over Nutch. A time limit is set for processing each user query. If the query processing time exceeds the limit, the query is terminated without providing any search results. The reason is that complex queries take longer processing time. These queries block other queries to be processed.

- **Project Crawling:** In order to find repository information for source code crawler, a custom crawler has been written in Python. It crawls the whole project in the web. Next, it finds the home page which contains metadata of the project such as license, documentations, etc.

- **Project Processing:** After getting the home pages of the crawled projects, web mining techniques are employed to extract project name, license and other useful metadata. All these information are stored into a database for delivering these to the users when they submit queries.

- **Project Searching:** An open source search application named Solr [94] is employed to provide project-centric search support. The Solr index schema is given in Table 2.8.

2.3.5 Large-Scale Code Search Infrastructure

Due to the large availability of open source code in the web, developers like to reuse these codes as much as possible. Developers need different tools to search and use

Table 2.7: Dependency Types Supported by Sourcerer [2]

Relation	**Description**	**Example**
INSIDE	Physical containment	java.lang.String
EXTENDS	Class extension	java.util.LinkedList EXTENDS java.util.AbstractSequentialList
IMPLEMENTS	Interface implementation	java.util.LinkedList IMPLEMENTS java.util.List
HOLDS	Field type	java.lang.String.offset HOLDS int
RETURNS	Method return type	java.lang.String.toCharArray() RETURNS char[]
READS	Field read	...String.<init>(java.lang.String) READS java.lang.String.offset
WRITES	Field write	java.lang.String.<init>() WRITES java.lang.String.offset
CALLS	Method invocation	...String.indexOf(int) CALLS java.lang.String.indexOf(int,int)
INSTANTIATES Constructor	invocation	foo() INSTANTIATES java.lang.String.<init>
THROWS	Declared checked exception	java.io.Writer.write(int) THROWS java.io.IOException
CASTS	A cast expression	java.langString.equals(java.lang.Object) CASTS java.lang.String
CHECKS	An instance of expression	java.langString.equals(java.lang.Object) CHECKS java.lang.String

reusable code fragments. In order to develop a code search tool, several challenges have to be faced. For example, crawling projects from the web, parsing source codes, storing in the database, searching over the code base, etc. Considering the challenges, a large scale code search infrastructure named Sourcerer [11] has been developed. In this subsection, the architecture of Sourcerer is discussed.

Table 2.8: Solr Index Schema for Projects in Krugle [3]

Field Name	**Description**
name	The "user friendly" project name
exactname	A unique name for the project, used as a key
description	A short description of the project
license	The open source license for the project
language	The programming languages used by the project
os	The target operating system(s) for the project
homepage	A URL to the home page of the project
filesurl	A URL to the top location of the code files in the project
metrics	Various code metrics extracted from the data of the project
boost	A floating point number used to alter search results

Functionalities of Sourcerer

Sourcerer is a web-based code search infrastructure that indexes entities in open source projects. It performs three basic functionalities which are as follows.

(a) **Collection and Storage:** Sourerer crawls web pages on the web and collects source codes. It also crawls metadata and stores these in the database.

(b) **Analysis and Indexing:** Sourcerer extracts both lexical and structural information of the crawled entities.

(c) **Search and Retrieval:** Lucene has been used in Sourcerer to construct index of the entities. Sourcerer accepts both free text and boolean query and retrieve result from the index.

Data Model of Sourcerer

In order to know the basic source code elements that Sourcerer stores in the database, the relational data model is shown in Figure 2.6. According to the figure, there are five models - Project, File, Entity, Comment and Relation. The model *Project* stores all the projects found in the Sourcerer repository. A project comprises a collection of source files and library files. Since Sourcerer supports are

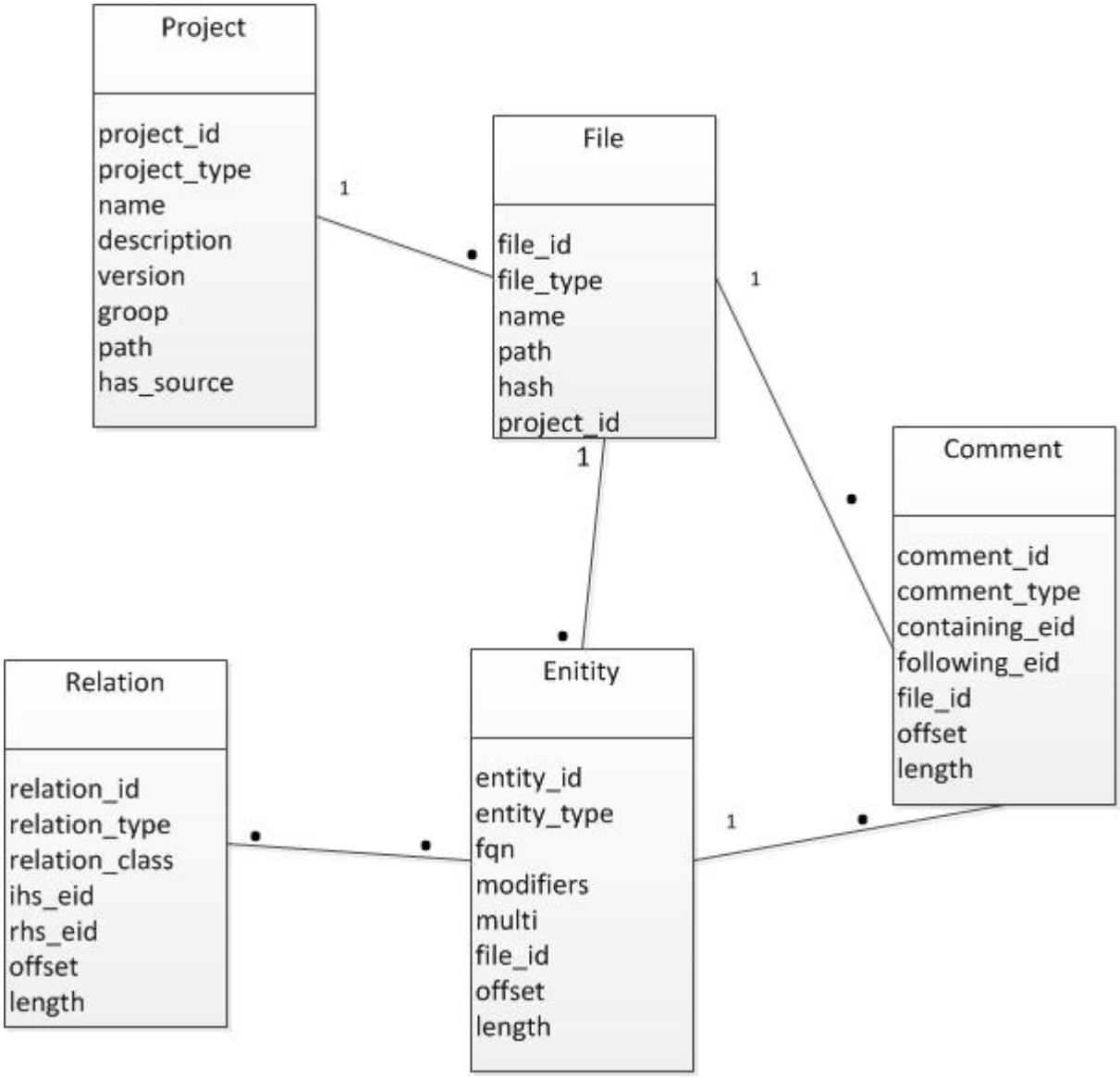

Figure 2.6: Data Model of Sourcerer [2]

confined in Java projects; Java source files, class files and jar files are stored in the model *File*. A source file or class file contains a collection of entities like method, variable, field, class, etc. A list of entities that Sourecerer supports, is given in Table 2.9. The Model *Entity* is used to store these entities. Dependencies among the entities are stored in the model *Relation*. Various dependencies are stored into *entity*, which are shown in Table 2.7. Comments found in the Java source codes are stored in *Comment*.

Table 2.9: Entity Types Supported by Sourcerer [2]

PACKAGE
CLASS
INTERFACE
ENUM
ANNOTATION
INITIALIZER
FIELD
ENUM CONSTANT
CONSTRUCTOR
METHOD
ANNOTATION ELEMENT
PARAMETER
LOCAL VARIABLE
PRIMITIVE
ARRAY
TYPE VARIABLE
WILDCARD
PARAMETRIZED TYPE
UNKNOWN

Query Support in Sourcerer

Sourcerer supports two types of queries - free text and interface driven. In free text query, a set of keywords is obtained from the users to retrieve relevant source code. This type of query is commonly performed by the developers when they have no or little knowledge about the interface of the desired components. Developers can also provide the signature or interface in the query and Sourcerer retrieves based on that. An example query is shown as follows.

return_type: *(String)* AND ***keywords***: *(merge)*AND ***parameter_types***: *(String[])*

For the query, Sourcerer searches in the index to retrieve the methods which take an array of *String* as arguments and return a *String*. In addition, the methods have *merge* keyword in the names. This type of query facility assists the developers to retrieve their desired codes that contain the specified interface.

2.4 Summary

Code reuse is considered as the main motivation for code search. Due to the large availability of the open source software projects, developers perform code search for component or snippet retrieval. This chapter presents the emergence of code search in software development. Source code file and normal document file are different because a source file is more structured than a normal document file and it contains little textual information. To put special consideration while developing code search engine, this chapter discusses the differences between normal document search and code search. Since the basic architectures of code search engines are built on Information Retrieval, a notion on IR is provided in the chapter. The demand of code search tools is increasing day-by-day. To have an overview about the general architecture of code search engines, two large scale code search infrastructures named *Krugle* and *Sourcerer* are described at the end of the chapter. The next chapter presents a brief literature review on code search.

Chapter 3

Literature Review on Code Search

Reusing existing code fragments reduces development time and effort [95]. For this reason, searching for reusable code snippets during software development has become a common task among the developers [39]. Various techniques have been proposed in the literature to improve the effectiveness of code search engine in terms of recall, precision, query successfulness, etc [14]. These techniques can be broadly classified into four categories which are Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Semantic Based Code Search (SBCS), and Test Driven Code Search (TDCS). Significant works related to each category are discussed in the chapter.

3.1 Keyword Based Code Search (KBCS)

In KBCS, source code is considered as plain text document where traditional Information Retrieval centric approaches are employed to index the code and query over the index [8]. Besides, other metadata such as comments, file name, commit message, etc. are used to retrieve relevant code fragments from a repository of

source codes. JSearch, Thesaurus Based Query Expansion, Sourcerer, UPMiner are significant works related to KBCS. These works are described as follows.

3.1.1 JSearch

Developers often use common libraries or frameworks to develop particular features of software. In order to accomplish a task, a sequence of operations is required to be invoked as defined in the library. Developers need to go through the documentations of the library to understand such invocation sequence. In a large organization, it is possible that some developers are working on the same libraries. One may not be aware that others are also using these libraries. One of the developers may gain insight about the usage of the libraries and accomplish the development of specified task. One the other hand, other developers may be inspecting the documents to know how to use these libraries. Additional time and cost are incurred due to the lack of communication.

In order to alleviate the problem, a tool named JSearch has been proposed by Renuka [6]. A repository of source code is maintained which contains all the codes written by the developers. The tool parses the source code and extracts terms from the class name, method signature, field name, package name, etc. A search index is constructed where extracted terms are used to index the source code. In order to optimize the index size, stop words like a, an, the, to, etc. are removed from the term list. The tool takes a set of keywords as query from the developers and searches the index to retrieve relevant codes that match against the query terms.

The tool was evaluated in terms of performance, scalability and relevance of the retrieved results. On an average 12ms is required to index a source code file and additional 30ms time is needed to preprocess each file before indexing. To assess the relevance of the retrieved code examples, a sample of 500 user queries for a period of 4 months was collected. These queries are broadly classified into

three categories-

Type 1: The developers have no idea of what libraries should be used to solve their problems

Type 2: The developers have queries related to the runtime environment

Type 3: The developers know what libraries should be used but they need example codes that demonstrate how to use these libraries

According to the result analysis, 23% of the total queries were handled by the tool by giving relevant search results. It could not retrieve relevant example code for other queries because these were either related to Type-1 or Type-2 queries.

The tool seems to have the capability of constructing index for large scale source code repository. However, it cannot retrieve all the code snippets that implement the same feature but contain different keywords. This is because it does not check feature-wise similarity to detect common terms for these fragments. Thus, the recall of the tool is reduced.

3.1.2 MAPO

In order to improve the productivity during software development, developers like to use the APIs exposed by class libraries or frameworks. It is a common form of software reuse. To be familiar with a framework, documentations are provided to the developers. In most of the cases, instructions to use the library or framework are not well documented. Developers often find the frameworks difficult to reuse for complex and insufficient documentation. To know the usage of a framework, developers search in the general search engines like google with the hope of finding example codes. However, a large number of documents and source codes are retrieved that makes it difficult for the developers to find their desired search results. As a result, developers have to put additional effort and time to reuse the desired framework.

A technique named MAPO [72] has been proposed in the literature to mine API usages from open source repositories. The technique has been devised to satisfy four objectives which are as follows.

- The technique should be able to extract API usage information from the source files of a project
- The technique should be able to infer frequent API usages including method call sequences
- Frequent API usage patterns should be mined by the technique
- The technique should be able to produce a short list of relevant frequent API usage patterns

To satisfy the aforementioned objectives, the technique is decomposed into five components and these are *a source code search engine, a source code analyzer, a sequence preprocessor, a frequent sequence miner,* and *a sequence post processor.* The technique takes class name, method name, input types, output types, etc. as a query. Based on the information, a source code engine retrieves the source files from the repositories. A source code analyzer parses the source files to obtain invoked API statements and invocation sequences. The sequence preprocessor module takes the invocation sequences and removes the entities that do not contain the query terms. Next, the frequent sequence miner uses BIDE algorithm to mine succinct API call sequences or patterns.

The technique was evaluated on a set of queries for two frameworks named BCEL and Javassist. It parsed 36 files from BCEL and identified 1087 invocation sequences. The number of sequences was reduced to 186 by preprocessing the calling sequences. Finally, the technique retrieved 8 frequent usage patterns for BCEL. 50 files were parsed by the technique to retrieve 828 invocation sequences in Javassist. 23 frequent patterns were mined by preprocessing and employing

BIDE algorithm. The result analysis depicts that the technique is practical for providing informative and succinct API usage patterns. However, the technique retrieves source file through different code search engines. Traditional IR based approaches are followed by the search engines that consider source code as a plain text document. Thus, if an API contains terms that are irrelevant to its implemented task, the API will not be retrieved.

3.1.3 Thesaurus Based Query Expansion

The effectiveness of a code search engine depends on the accuracy in representing the user-needs into query terms [56]. Proper transformation of information-needs into query terms helps to retrieve more relevant codes. Most of the code search engines provide a single text box to obtain a set of keywords that maps user query [26]. These keywords are matched against the collection index to retrieve relevant code snippets or software artifacts. However, if user provided keywords do not express the desired search topic or context properly, search engines fail to retrieve many relevant codes [96]. Sometimes, many irrelevant code fragments are shown as top ranked search results by search engines due to using irrelevant keywords as query terms. This problem is commonly known as vocabulary mismatch problem in IR [56]. The problem is also relevant to code search as keywords are used to define user query [20].

Lemos et. al. proposed a thesaurus based query expansion technique to handle vocabulary mismatch problem in code search [20]. The technique performs two types of expansion which are query expansion and type expansion. Query expansion is applied on the textual part of the query which does not rely on the syntax of the program languages. For each query term, synonyms and antonyms of the term are obtained for query expansion. Synonyms are appended to the query term using Boolean OR operation and antonyms are joined using Boolean AND operation where NOT is added to each antonym. Type expansion is carried out on the

syntax specific keywords in the query, for example, data type of a parameter in a method signature. Language specific thesaurus is built for the expansion, and related keywords are joined with the query terms through Boolean OR operation. Finally, the expanded query is submitted to the search engine to retrieve more relevant code fragments.

A repository of 2,000,000 Java methods was selected for the evaluation of the technique. 16 different auxiliary functions were randomly gathered from the literature and 36 subjects were employed to identify the methods relevant to the auxiliary functions from the repository. A set of queries was collected from the subjects where each query represented one of the auxiliary functions. For each query, the relevance of the retrieved methods was judged whether these implement the intended auxiliary function or not. The use of type and query expansion improved recall by 30%. On an average, 43% improvement in the number of retrieved relevant functions was shown by the technique.

The technique improves recall if an user query is ambiguous and does not contain proper terms to represent the user needs. However, both query expansion and type expansion cannot retrieve more relevant methods if method name does not comply with the implemented functionality. The reason is that terms are extracted from method name for indexing and if method name is inconsistent, extracted terms will be inappropriate.

3.1.4 Sourcerer

Developers perform code search for various reasons such as reusing existing code fragments [28, 74], understanding the usage of a library function [69], knowing API usage patterns [72], obtaining example code to implement feature, etc. Search intent differs from one developer to another developer due to different development contexts. For example, one developer may be interested in example code snippets that implement a particular feature whereas another one may be interested to

know how to use a library. It is not cost effective to create multiple search engines to fulfill different types of the user-needs. Again, it is also difficult to manage when code base increases rapidly with the passage of time. So, a single infrastructure needs to be developed which enables the developers to obtain search results as per their search intents.

Sourcerer [11] is a large-scale code search infrastructure that stores the source code in a more structured way. It automatically collects and stores the code in a repository with a standard directory structure. It performs in-depth structural analysis of the source code by identifying dependencies. Structural information like which classes are under which package, which class contains which methods and fields, which methods are invoked by a given methods, inheritance hierarchy, etc. are stored in a database. Dependencies like used libraries, invoked methods defined in other classes, inherited fields and methods, etc. are also saved in the database. Sourcerer constructs index of source code under different granular level such as class level, method level, package level, and so on. To build the index, keywords are obtained from the source code, and these are stemmed and tokenized. A search interface is provided by which user can give query in different formats such as signature of method, only keywords, API name, etc.

Although these techniques use both types of information to fetch more relevant code fragments, these cannot retrieve feature-wise similar code blocks simultaneously. The reason is that all these information are stored following IR-based indexing mechanism, and no checking is performed to index similar code snippets under the common proper terms. Here, code snippets are similar in the sense that these do the same task but may contain different terms from each other.

3.1.5 UPMiner

Application Programming Interface (API) is one of the most common forms of software reuse [51, 97, 98]. Usually, when developers are more interested to reuse

existing libraries or components rather than the implementation of the libraries, API is one of the most common choices to the developers. API comprises functionalities that are exposed to the developers to reuse these for developing or accomplishing a particular task. Often a function defined in an API needs to invoke another function to perform a certain task. Such sequence of function invocations is known as API usage pattern. In order to gain insight into different usage patterns, developers look for documentation, example codes and other resources. Different automated tools are found in the literature that cooperates the developer to obtain API usage patterns.

Documentation is considered as one of the most important sources of information to understand API usage patterns. However, in most of the cases, API documents do not contain detailed and up-to-date information. Even, most of the APIs do not have concise documentation due to poor project management. In addition, these documents are not updated with the passage of time and changes of the API libraries [99]. So, developers find very little information about API usage patterns which contributes a little to the developers for reusing API functionalities. The next source of information is collection of example codes in which API functions are used for the implementation of different tasks or features. This source can be utilized to mine usage patterns of API. However, it is challenging to obtain meaningful and useful patterns from this source. Quality of mined patterns is very crucial as the success of usage pattern mining technique depends largely on the quality which is defined by the user needs to complete a task. So, all these issues lead to the research question which is how to mine quality API usage patterns from large code base so that these will help developers to reuse API methods quickly in practice.

Two novel metrics were proposed by Tao et. al to measure the quality of the mined usage patterns from large scale code base [100]. One of these was succinctness which defines that mined patterns should be succinct as developers

are more interested in small usage patterns rather than inspecting the redundant patterns. Another metric was coverage which depicts that obtaining all possible usage patterns including less usage patterns. These two metrics jointly help to evaluate the mined patterns quality of different techniques. On the other hand, a technique named UP-Miner was also proposed to obtain succinct API usage patterns from large scale code base covering all possible usage patterns. The technique first gathers a set of call sequences that use API methods. It then applies clustering technique based on the similarity among the sequences. A minimum support threshold is defined while clustering API method invocation sequences. The reason behind applying clustering before mining usage patterns is that less frequent patterns that do not support minimum threshold for clustering are mined out before. An n-gram based similarity score is presented in this paper to calculate the similarity score between two sequences of method invocations. Two intuitions are associated with the similar sequences identification. Firstly, two sequences are similar if they share high percentage of items between their n-gram sets. Secondly, two method sequences are highly similar if they share longer consecutive sequences rather than shorter consecutive ones. After that, frequent closed patterns are mined using BIDE algorithm. Sequences from different clusters may be similar to some extent, for this reason, one time more clustering is performed to find all similar sequences that represent API usage patterns. At last, search-based approach is used to find optimal number of usage patterns with the consideration of high quality in terms of succinctness and coverage.

In order to evaluate the proposed technique, it was implemented in form of a software tool. An existing technique named MAPO [72] was used for comparative result analysis. 20 popular .NET API methods were selected as experimental dataset. Besides, two metrics named purity and inverse purity were also adopted in the experimental analysis. The result analysis shows that UP-Miner outperforms MAPO on 17 API in terms of F-measure. With an average of 11.92% UP-Miner

performs better than MAPO in duplicate patterns reduction. The reasons of such outperformance are two steps clustering adoption, similarity metric formulation and mining similar closed patterns.

UP-Miner is a promising technique in mining API usage patterns from large code base. The technique showed better performance than existing technique in terms of high coverage and succinctness. However, it uses keywords from the API methods to construct index without considering similarity among these methods. Thus, the recall is reduced in retrieving relevant API methods.

3.1.6 XFinder

Frameworks are required to be extended to incorporate new changes and features. In order to extend or customize a framework, developers look for tutorials or documentations that provide high-level description about the framework. However, developers are more interested in workable example codes than documentations. They can obtain a clear and practical idea about the framework from implementation point of view. Again, codes shown in the tutorial, can also be obtained from the existing projects in which the framework has been used. Finding an extension example within an existing project requires those pieces of codes that implement the extension. It is difficult to find all the pieces of code because these might be located in different locations of the project. Some pieces of codes might be shared among several extensions. Some might be intermixed with other codes. Even, if a developer finds the important code elements that implement the framework extension, it is difficult to find the relationship among the elements. It is also hard and time-consuming to know the responsibilities of the elements. The reason is that manually inspecting a large code base is not feasible at all. So, it is required to automatically find the code elements that will assist the developers to understand the framework extension.

A technique named XFinder [71] was proposed to find example framework ex-

tension codes from a code base. The technique uses the initial documentation of a given framework to identify the concepts. The technique uses an existing technique named Mismar [101] to identify concerns of a given extension documentation. These concerns are used by the technique to construct concern templates. The technique then matches these templates in the code base to find the similar instances of the extension. Code fragments that follow the templates are retrieved as example code snippets for the similar extension of a framework.

An expert study on the Eclipse text editor was conducted to evaluate the effectiveness and applicability of the technique. XFinders was run to find the example code snippets on Eclipse text editor framework extension. The obtained results were then compared to the expertise of the developers who performed the extension. According to the study, XFinder correctly locates 93% of the implementation pieces of five eclipse text editors.

The technique helps to locate example code snippets in a code base. It depends on the initial documentations of a framework extension. The accuracy of the technique depends on the appropriateness of the initial documentation. However, developers do not like to write document for an extension [92]. Again, writing document using Mismar requires extra effort to the developers.

3.2 Semantic Based Code Search (SBCS)

Since the amount of open source code is increasing day by day, it is thought that a significant amount of code that is written today, has already been available on the internet [102]. However, reusing the existing code often does not directly meet user needs properly. Again, the code needs to be modified to accomplish the intended task under the new development context. Understanding the existing source code and modifying to work in the development environment are time-consuming and tedious. So, semantics of the source code needs to be matched with user query

so that the effort is minimized to make the code workable. Several techniques are available in the literature to retrieve semantically relevant code fragments from a given code base. These are discussed in the following subsections.

3.2.1 S6

When implementing a feature, developers look for existing codes that are consistent with the feature instead of writing the codes from scratch. In most of the cases, existing code snippets do not implement the exact feature. Few code fragments are found that are similar to the desired feature but some of these may too difficult to modify. Again, some fragments may be hard to understand or some may have higher complexity in terms of time and space. A significant portion of time is spent by the developers to understand the semantic of the code before reusing it.

In order to identify semantically relevant code fragments, Steven P. Reiss proposed a technique named S6 [17]. It takes a set of keywords and a collection of test cases from the user to represent the search intent properly. Two types of keywords are required to be provided by the user, one of these is component name and another is type of the component with interface description like class, method, exception, test case, etc. Using these keywords, relevant components are retrieved as initial search results. After that, the components are transformed to match with the interface description. If such transformation is not possible, the respective component is removed from the search results. The filtered results are checked against the given test cases and components that successfully pass the test cases are considered as final search results. The technique ranks the search results based on the different metrics such as cyclomatic complexity, line of code, coupling, cohesion, and so on.

In order to evaluate the proposed technique, a tool was developed using Java programming language. The tool was run to find relevant code fragments against

different tasks such as tokenizing string, converting to roman, concatenating string, sorting array, generating prime numbers, etc. On an average time, the system took 120ms for single thread time and 47ms for eight thread time. Moreover, the set of transformation rules were found powerful but insufficient for some cases like handling generic types, reflection, and so on. Although the proposed approach performs well in terms of precision, recall is reduced since proper terms are not determined while indexing feature-wise similar codes. So, some semantically similar code fragments cannot be fetched due to indexing these fragments under inappropriate terms.

3.2.2 XSnippet

"Develop by example" is a common practice seen among the developers while developing software. It guides the developers to accomplish tasks through using existing libraries or frameworks. The importance of example code increases with the complexity of the framework or library being used and experience of the developer with that. So, example codes are provided with the library or framework to ease the usage of it. In fact, it is said that the availability of good examples is considered as the key factor for the successfulness of a library. With the passage of time, the amount of example codes is increasing rapidly. It is difficult and time consuming to find relevant example codes by manually inspecting the source code files.

To get example code, Niyana proposed a technique named XSnippet [18]. The technique finds example codes that convert one object type to another as specified by the user query. Firstly, it parses the source code of a given code base and constructs Abstract Syntax Tree (AST) for each source code file. After that, it creates graph from source code by adopting code mining algorithm. The graph represents data flow within the corresponding source code. Each node in the graph denotes the type of variable or field found in the code and an edge from

A $\rightarrow$ B expresses that A is converted to B. Moreover, user query is defined by providing input type and output type. For a user query, all the generated graphs are searched to find those code fragments that convert the input type into the output type. The resultant code fragments are delivered to the user as the search results.

In order to evaluate the proposed technique, a repository of 2,000 Java class files and 22,000 methods were extracted from two standard eclipse plugin named org.eclipse.jdt.ui and org.eclipse.debug.ui. 17 object instantiation tasks were selected for the assessment. The result analysis shows that the technique outperforms for all the tasks in comparison with an existing technique named Prospector [69]. The reason is that Prospector query evaluation is limited to the lexically visible types within the class boundary and it does not check the variables or fields declared in the super class.

In XSnippet, developers have to provide exact input type and output type for getting example code blocks. Otherwise, it cannot retrieve code fragments that may satisfy user needs. However, according to the searching behavior, developers are more interested in using keywords rather than concrete data types to define their queries [26].

3.2.3 Portfolio

Different studies show that programmers are more interested in the definition and usage of a function rather than variables, statements, or arbitrary code fragments [26, 103, 104]. In order to find the solution of a particular task, programmers search for functions along with their usage contexts. In most of the cases, a sequence of functions needs to be invoked to complete a task instead of a single function. Traditional code search engines provide little support for this purpose. These engines cannot retrieve the usages of the functions with invocation sequence at all due to not considering usage context. As a result, reusing source code becomes

difficult to the programmers.

Collin et al. proposed a technique named Portfolio to retrieve source code demonstrating the usage of relevant functions and the invocation sequence [105]. The technique employs surfing behavior and related concept in source code. Here, surfing behavior is determined by parsing the source code, identifying function calls and constructing call graph. On the other hand, relevant concepts are obtained from function name, invoked method name, declared fields and comments. User provides keywords in the form of a query and these keywords are matched to the terms that define concepts to retrieve relevant functions. Search results are shown graphically so that user can perceive the usage context of a function through browsing the graph.

A large code base named FreeBSD was used to evaluate the effectiveness of the technique. There were 270 Millions Line of Code in the code base. Two existing popular search engines named Google Code Search and Koders were also included in the experiment for comparative result analysis. Besides, 49 professional C++ programmers were employed to judge the relevance of the results retrieved by Portfolio, Google Code Search and Koders. The results showed with strong statistical significance that users found more relevant functions with higher precision with Portfolio than with other two engines. However, it cannot retrieve similar codes at all if semantic information denotes different meaning to the implemented functionality. This is because of having inappropriate names or keywords in the method signature or invoked method name.

3.3 Test Driven Code Search (TDCS)

In TDCS, test cases are used to define the behavior of the desired code fragments that developers are looking for. Usually, test cases provide instant feedback about the suitability of a particular code fragment in the local context [106, 107]. A

test suit also defines the dynamic behaviors of the code fragments. In this type of technique, test case is used as search interface and retrieved code snippets are run against the test case to provide relevant and reusable code fragments. CodeGenie [15], Merobase [16] and Exemplar [45] are three predominant techniques under this category and these are discussed in the following subsections.

3.3.1 CodeGenie

As the availability of open source code is increasing day-by-day, currently developers are more interested to reuse source code rather than writing the same code again and again. Around 30-50% of the total development time is spent by the developers for finding the reusable code and plugging into the development environment [92]. The reason is that retrieved piece of code needs to be manually inspected to determine its functionality, quality, dependencies, and so on. Again, manually adapting code fragments in the local development context is also error-prone and tedious task.

Lemos et al. introduced the use of test case as an interface for automating code search and reuse, and they developed a tool named CodeGenie [15]. Test cases are used for two purposes, one of these is to define the behavior of the desired code fragments and another is to check the suitability of the retrieved code fragments in the development context. The proposed technique supports method level searching. It takes method signature as a query from the test cases written by developers. It uses Sourcerer infrastructure to retrieve relevant functions against the query. Next, all the test cases are executed for each retrieved method. Resultant methods are ranked based on the number of test cases successfully passed.

In order to evaluate the performance of the proposed technique, 34 examples of auxiliary functionality were selected in the experimental study. 41 subjects were employed to judge the relevance of the retrieved code snippets. Besides, Google Code Search and a manual approach were included in the study for the

comparative result analysis in terms of reusing code snippets. The comparative study showed the evidence of feasibility and better performance of CodeGenie than other two approaches in finding and reusing code fragments automatically.

The technique seems to increase precision, but it produces low recall. The reason is that it performs keyword matching to fetch methods from index without justifying the appropriateness of the keywords. Again, keyword matching cannot retrieve syntactically different but feature-wise similar codes as stated earlier.

3.3.2 Merobase

One of the major challenges in software reuse is to deliver high quality search results with high precision and semantic recall [5]. A promising approach named CodeGenie which is described earlier produces high precision since it employs test case to obtain behavioral requirements of a developer. However, it suffers from low recall when relevant components are semantically similar but syntactically wrong. Test cases defined in the query may not pass for the syntactically incorrect components. As result, many semantically relevant code fragments cannot be retrieved. Such reduced recall limits the scope of reusing existing source code.

Usually, retrieved methods may not pass corresponding test cases due to different order of the parameters, return type or parameter type. To resolve these issues, Janjic et al. proposed a technique named Merobase. The technique refactors the code to adapt with the program context [16]. It applies every possible adaptations like reordering parameters, using super type or sub class type of a given return or parameter type, converting primitive type to reference type or vice versa, etc. Thus, it improves TDCS by finding more relevant methods.

A prototype was implemented using Java programming language but no experimental analysis was performed to assess its usability and performance. The motive of the technique is to retrieve semantically relevant code fragments through adapting source code under the local development context. Still, it seems to pro-

duce low recall because it first retrieves initial search results based on the keywords specified in the test case. In addition, it does not check the similarities among a given collection of methods while constructing an index. Similar methods are not indexed under common terms as well.

3.3.3 Exemplar

Software developers face many challenges while searching for reusable source codes in a large code base. Usually, different documentations like software architecture, component description, requirements document, etc. provide the high-level descriptions of a software project. Matching these high-level descriptions to the low-level implementation is a challenging task to retrieve relevant codes. Again, user queries are provided in a set of keywords that represent the high-level description of the desired software project. Relevant codes cannot be retrieved when the high-level intents mismatch to the low-level implementation. This problem is commonly known as *concept assignment problem*. Different search engines have been developed to solve the problem. The search engines extract the textual keywords from the comments, variable names, function name, documentations, etc. The high-level query is matched to these keywords to retrieve relevant codes. However, many repositories contain poorly functioning projects. So, when source codes from these projects are retrieved, developers can hardly reuse these. If a user query matches to the description of a project, it does not guarantee that the project is relevant to the user query.

In order to ensure the relevance of the retrieved codes, a technique named *Exemplar (EXEcutable exaMPLes Archives)* [45] has been proposed in the literature. The technique finds highly relevant executable applications from a given repository of projects. In order to index the applications, the technique takes three types of information. It takes textual or lexical words from the description of the applications. Next, it extracts API names that are invoked in the applications.

At last, it determines the data flow among the APIs. When a user query is given to *Exemplar*, the technique first matches the query terms to the description of the applications. A set of applications is retrieved and API names of the applications are matched to the provide query terms. The API matching scheme provides a collection of relevant APIs. A data flow relationship is established among the APIs to retrieve all the relevant codes requires for the execution. These APIs with the dependent codes are then delivered as the search results.

A case study was conducted to evaluate the effectiveness of the technique. 39 professional programmers were selected to evaluate the technique. The subjects were given a set of tasks. They were asked to solve these tasks using three different search engines which were Sourceforge, Exemplar without data flow and Exemplar with data flow. According to the comparative result analysis, subjects found Exemplar more effectiveness than Sourceforge. They put less effort to find relevant codes and reuse these in comparison with Sourceforge. Another finding was that using API names for index construction increases the matching probability between query terms and index terms. In addition, retrieving relevant codes based on the data flow graph ensures the executability of the codes.

Exemplar is an effective technique for retrieving executable APIs from a code base. To check the executability, the technique constructs data flow among the APIs. It also relies on the textual keywords on the API names and descriptions of the application. It uses these keywords with checking the relevance during index construction. If an API name is not relevant to the implemented feature, the technique cannot retrieve that API. As a result, the effectiveness of the technique will be reduced.

3.4 Interface Driven Code Search (IDCS)

IDCS helps the developers to define their queries in a more structured form rather than just a set of keywords joined by boolean expression. For example, if a user searches for a method that returns the maximum between two numbers, the corresponding query in IDCS is "return-type:int AND method-name:max AND parameter:int AND int". Some significant works in IDCS are explained as follows.

3.4.1 Signature Matching

Reusing methods or functions of existing projects or libraries is one of the common forms of software reuse. A method performs a specific task, and a set of methods jointly implements a particular feature. With the increasing popularity and availability of open source projects, developers often look reusable methods in order to develop a solution of a particular task. If the existing projects are developed by assuring quality properly, it is not required at all to check the quality of the same methods collected from these projects. Reusing existing methods not only saves significant amount of time and cost but also ensures the quality of the software.

Signature matching was the first proposed IDCS technique to find relevant functions within a software library [108]. The approach crawls all the methods in the library, and uses signature of each method for indexing. User query containing method interface definition (that is signature) is executed to retrieve relevant functions from the index. Method interface definition includes the method name, list of parameters with corresponding data types and return type.

The technique was evaluated on a repository of twenty different software libraries. A set of real life user queries was selected where each query denoted the desired method name, parameters and return type. According to experimental results, the technique retrieved relevant methods for 78% user queries. However, it could not retrieve any relevant method for the rest of the queries because there was

no method in the repository that match to the specified signature or incorrectly defined data types.

The technique can retrieve relevant method if the user query is correctly defined. However, it loses recall if code base contains methods that are semantically similar but contain different names in the method signatures. This is because it takes terms from the method name defined in the query, and retrieves methods that match with the terms and signature.

3.4.2 Strathcona

One of the most popular forms of software reuse is framework or libraries. Using the APIs exposed by the libraries is challenging to the developers who are not familiar so much with those. It becomes very difficult to learn the complex object-oriented frameworks. In order to know the usage of a framework, four sources of information are provided to the developers. These are as follows.

- High-level documentation
- Code-level documentation such as API details
- Hand-crafted examples that illustrate how to use specific parts of the framework
- The source code of the framework itself, including embedded comments.

In most of the cases, developers find little information in the documentations. Sometimes, documents are outdated or incomplete. Hand crafted examples are so sparse in content that developers can hardly find their desired example codes. It is said that one of the best ways to understand a framework is to see the sample code snippets in which it has been used. To know the usage of a framework, a new developer approaches to another developer who is experienced in using the framework. The experienced developer finds some example codes that may help to

understand the usage of a particular APIs. However, finding such example codes manually in a large software system is a tedious task. It induces additional time and effort in software development.

In order to find the example codes of a framework, Holmes et al. proposed an automatic technique named Strathcona [70]. The technique takes the interface descriptions from the structural contexts of the API. For example, the structural contexts of a class are obtained from the class name, parent class name, fields, methods, package name and etc. The structural contexts for a method are collected from the signature of the method, invoked methods and so on. Using the structural information, the technique searches in a code base and finds relevant code snippets. Keyword based searching scheme is used to identify relevant code fragments. At last, search results are displayed graphically for enabling the developers to navigate the usage examples of the desired API. The navigation scheme assists the developers to find the invocation sequences of different APIs.

To evaluate the proposed technique, a plugin was developed for Eclipse integrated development environment. A control experiment was conducted to know whether the use of structural context assists to retrieve useful example code snippet. Two subjects were selected for the experimental. They were given four tasks to implement in Java programming language. Each task was related to writing plugins for Eclipse. For the evaluation, the source code of Eclipse were used as repository for Strathcona. The repository comprised 1.5 million lines of code. Subjects were asked to use the repository for the plugin development. They successfully developed the plugin and found Strathcona as an effective technique to know how to use APIs of a library.

The technique helps the developer to locate example codes that demonstrate the usage of a framework. It matches the keywords of structural contexts to the textual words in the source codes. The retrieval effectiveness depends on the appropriateness of the keywords that define the structural context. If irrelevant

keywords are used, the technique will miss many relevant example code snippets.

3.4.3 Prospector

Developers often prefer to reuse existing libraries or frameworks when implementing common software features. However, reusing the libraries or frameworks is difficult to the developers for two reasons. The first reason is that a software framework exposes many functionalities in the form of APIs. For example, the Java standard library J2SE provides a wide range of reusable components. The library contains thousands of classes and more than 21,000 methods. It is very difficult for a developer to remember all these components. Another reason is that libraries are made flexible to increase maintainability and extensibility in the long run. Functionalities of the libraries are decomposed into multiple small composable units to increase reusability. So, when a developer uses the libraries to implement a feature, several units are needed to be integrated in the development context. This imposes the developers to know the units before using the frameworks.

To assist the developers reusing APIs exposed by the software libraries, an automatic technique named Prospector [69] was proposed in the literature. The technique finds relevant code fragments about the usage of the APIs so that the developers can get idea about how to use and integrate those APIs into their development contexts. The technique takes two inputs from the developers - τ_{in} and τ_{out}. Here, τ_{in} denotes the input class name and τ_{out} represents the output class name. That is, the technique retrieves code snippet that converts τ_{in} to τ_{in}. To find the relevant code fragments, the technique synthesizes the signatures of the APIs. The input data types and the return data type can be obtained from the signature of an API. Using these information, the technique constructs a graph by source codes in a code base. Each node in the graph represents the data type. An

edge from node a to b represents that a is converted to b by an API. The name of the API is used as a level of the edge. When a user query is given in the form of (τ_{in}, τ_{out}), the generated graph is traversed to find all the paths from the node τ_{in} to τ_{out}. For each path, API names in the edges are extracted and combined to deliver as the search result. Although all the search results may not be relevant to the developers, the technique reduces the search space by providing short list of code snippets.

For the experimental analysis, a prototype of the technique was implemented in the form of Eclipse integrated plugin. To test the technique, 20 real life programming problems were used in the experiment. All these problems were related to the use of different APIs. For example, reading lines from an input stream, opening a named file for memory-mapped I/O, getting table widget from an Eclipse view, getting the active editors, etc. The technique identified the desired solutions for 18 out of the 20 problems. A case study was also carried out to evaluate the technique. In the study, four programming problems were given to a group of programmers. The subjects were asked to solve two problem using Prospector and two other problems without Prospector. They were also told that they could reuse source codes. On an average, the programmers solved the problems two times faster using Prospector. They reused code snippets while using Prospector. For other two problems for which they were not permitted to use Prospector, they wrote most of the codes by hand. Thus, the study justifies that technique helps the developers to reuse code snippets.

Prospector retrieves the example code snippets to convert one input type to another using APIs. It depends on the signature of the API to determine the input and output types. There exists many APIs in different frameworks or libraries that do not have any parameter, return type or both. The technique cannot construct the graph for these APIs. As a result, it cannot retrieve the APIs against the user query. Again, developers need to have prior knowledge about the input and

output types. They have to explicitly define the types in the query to retrieve relevant codes. However, it is difficult for the developers to define the queries if no documentation is available for a give software framework. Developers do not like to use explicit types in their queries to retrieve reusable codes [92].

3.5 Summary

In order to find reusable code fragments, four types of techniques have been proposed in the literature which are KBCS, IDCS, TDCS, and SBCS. KBCS uses IR-based approach to index and retrieve relevant codes, IDCS refines user query by defining required interface definition, TDCS and SBCS use test cases to get program semantics. All these techniques extract keywords from source code to generate terms, and index corresponding code against the terms. However, none of the techniques checks the appropriateness of the terms with respect to the implemented feature. As a result, the number of relevant codes retrieved is reduced due to indexing against improper term. If keywords found in a code fragment do not represent the implemented feature, matching probability of query terms with indexing terms is reduced. Moreover, if two or more code snippets implement the similar feature but contain different terms, existing techniques cannot retrieve all these code fragments simultaneously. The reason is that these are indexed against different terms. To improve effectiveness in code search, feature-wise similar codes should be indexed under common appropriate terms. However, determining the intent of a code block, and selecting proper terms for syntactically different but feature-wise similar codes are open research problems in the literature.

Chapter 4

Retrieval of Functionally Similar Methods

The effectiveness of a code search engine is reduced if it cannot retrieve code fragments that perform the same task but contain different keywords. As a result, developers may miss many relevant code fragments against their search queries. To satisfy the needs of the developers, code search engines should provide all the functionally similar codes despite having different keywords. For this, it is required to identify the implemented feature of a code snippet. Later, clustering needs to be carried out based on the functional similarity so that semantically relevant codes can be retrieved. Appropriate terms for each cluster are needed to be determined to construct index and increase the matching probability with the query terms. Considering all these issues, a technique named Similarity Based Method Finder (SBMF) has been proposed which improves the effectiveness in code search by indexing semantically similar methods under proper terms. A detailed explanation of the proposed technique is presented in this chapter.

4.1 Overview of SBMF

SBMF improves effectiveness in code search by retrieving all the functionally relevant methods. It determines the dynamic behavior that is, the implemented feature of each method in a given code base by executing the method. It clusters the methods based on the feature-wise similarities where methods in the same cluster are considered as functionally similar to each other. Relevant terms for each cluster are selected to index all the methods in the corresponding cluster against common terms. As a result, if a query term matches with the index term, all the methods of the respective cluster will be retrieved. Due to this indexing mechanism, if a method name does not represent the implemented functionality, it will also be retrieved. In order to have modularity while performing these tasks, SBMF is divided into seven steps. To understand the workflow of SBMF and interactions among its steps, an overview is shown in Figure 4.1. According to the figure, SBMF starts with parsing methods from a code base and ends with indexing functionally similar codes under proper terms. Now, each step of SBMF and its respective tasks are discussed as follows.

Step - 1: **Self Executable Method Generation:** This step is responsible to parse methods from a given code base and transform each method into self-executable code fragment that is program slice by resolving data, library, and function call dependencies.

Step - 2: **Method Signature Regeneration:** It may happen that existing signature of a self-executable method does not represent its functionality properly. In this case, signature of the method is regenerated by adding new parameters and/or return types.

Step - 3: **Input Dataset Generation:** For a given self-executable method, this step generates an input data set that covers all the branches and statements of

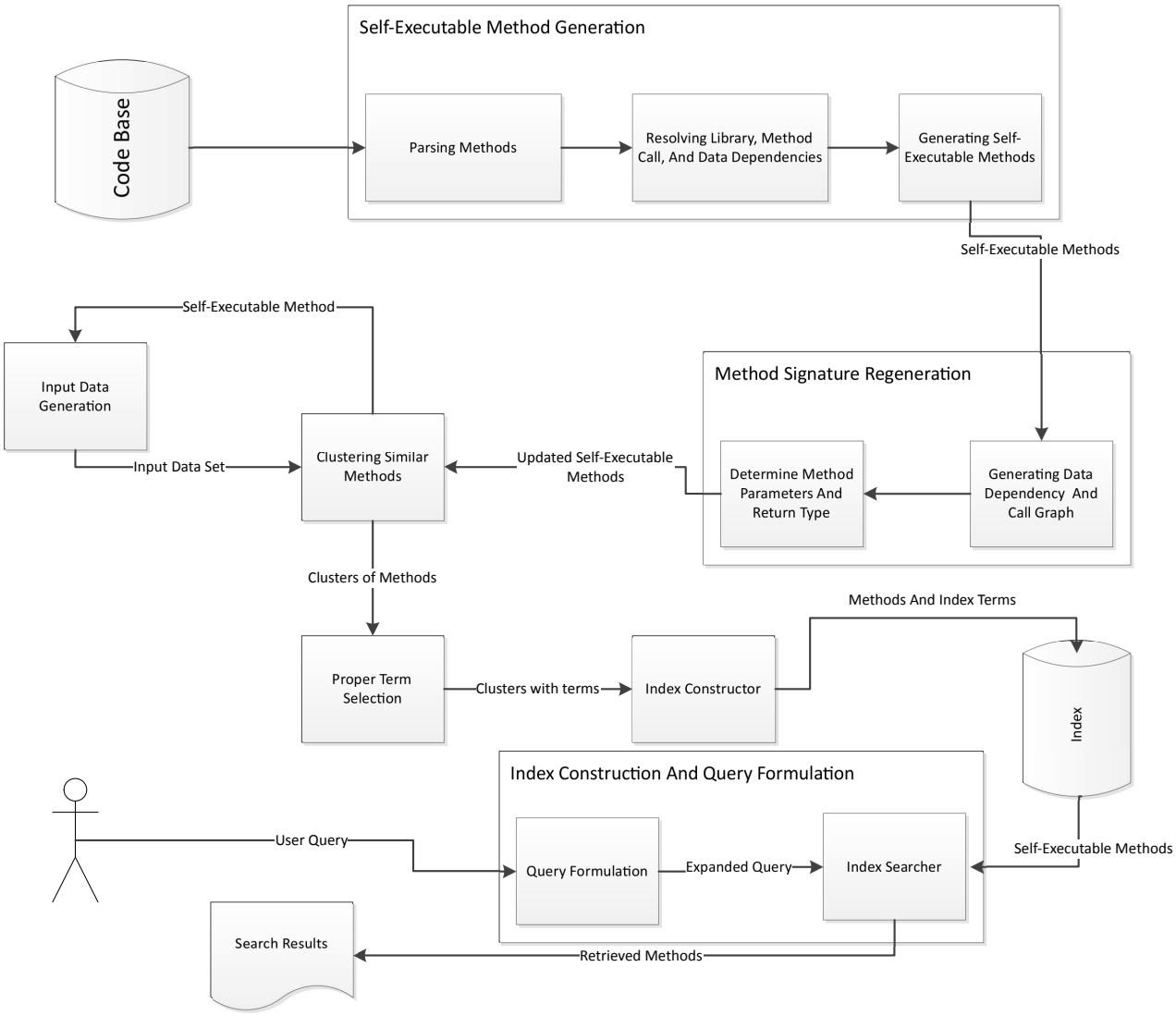

Figure 4.1: Overview of SBMF

the method. This dataset is used to determine the implemented feature of the method.

Step - 4: **Clustering Similar Methods:** Self-executable methods obtained from Step-2 (Method Signature Regeneration) are executed against the input set generated using Step-3 (Input Dataset Generation) for clustering the methods. Methods that produce the same output set for the same input set are stored in the same cluster as these are functionally similar to each other. In this step, the whole collection of self-executable methods is divided into a set of clusters. After that, these clusters are delivered to the next step to

select proper term for each cluster.

Step - 5: **Proper Term Selection:** For each cluster of methods, names of the methods are tokenized and stemmed to generate terms. Later, terms that are found in most of the methods are selected as index terms for the cluster.

Step - 6: **Index Construction:** In this step, all the clusters are indexed against the respective index terms. As a result, methods in the same cluster will have the same index terms. If a query term matches to any of the index terms, all the methods will be retrieved.

Step - 7: **Query Formulation:** User query is taken in the form of boolean query and it is expanded using log-mining based query expansion to solve vocabulary mismatch problem. Finally, the expanded query is submitted to the index, and relevant self-executable methods along with the dependencies are delivered as final search results.

This section provides a notion about SBMF along with its steps. The technique is further discussed elaborately in the following sections to know the underlying working mechanism and in-depth analysis of its steps correspondingly.

4.2 Self-Executable Method Generation

SBMF starts with parsing source files to identify all the methods in a given code base. In order to convert each method into self-executable method, three types of dependencies are required to be resolved. These dependencies are as follows.

- Method Call Dependency
- Data Dependency
- Library Dependency

4.2.1 Method Call Dependency

A method m may invoke another method m' directly or indirectly to perform its intended task. If m invokes m' in its body, it is said that m' is directly invoked by m. On the other hand, if m directly invokes a method x that calls another method x', x' is considered as indirectly invoked by m.

Definition 1. *A method m' is directly invoked by m if method call statement for m' is found in the body of m.*

Definition 2. *Consider an invocation sequence $[m_1, m_2, m_3, ...m_n]$ where m_i directly invokes m_{i+1} and $i = 1, 2, 3, ..., (n-1)$. m_q is said to be indirectly invoked by m_p if $(p+1) < q$ and $p, q = 1, 2, ..., n$.*

```
1 public String encrypt(String str) {
2 String encryptedStr = reverse(str);
3 return``%" + encryptedStr + ``%";
4 }
5
6 public String reverse(String str) {
7 str = getLowerCaseString(str);
8 String reversedStr = ```';
9 for (int i = str.length() - 1; i >= 0; i--) {
10 reversedStr += str.charAt(i);
11 }
12 return reversedStr;
13 }
14
15 public String getLowerCaseString(String str) {
16 return str.toLowerCase();
17 }
```

Listing 4.1: Example of Method Call Dependency

An example is shown in 4.1 to understand direct and indirect method call dependency. In the example, method *encrypt* takes a string *str* as input, and returns the encrypted form of *str*. Here, method *encrypt* directly invokes a method named *reverse* as it contains an invocation statement *reverse(str)* in its body. So, *encrypt* is directly dependent on *reverse*. Again, *encrypt* indirectly calls another method (*getLowerCaseString*) because *encrypt* has no method call statement for *getLowerCaseString* but *reverse* directly invokes *getLowerCaseString*. Thus, *encrypt* is indirectly dependent on *getLowerCaseString*. In order to make *encrypt* self-executable, method calls to *reverse* and *getLowerCaseString* are needed to be resolved. Otherwise, the method will not be compilable.

4.2.2 Data Dependency

A method may use one or more fields which are declared outside the method body. A field may also depend on another field for its initialization. Again, a method may use a field through its invoked method. Considering these cases, data dependency for a method is defined as follows.

Definition 3. *A method m is said to be dependent on a field f if any of the following cases satisfies-*

(1) the method body contains a statement that manipulates f

(2) any variable inside the method body uses f

(3) another field f' depends on f and f' is used in the method body

(4) f is used by its invoked method

```
String extension=``.exe";
String filePath=``\customdir\custom.exe";
FileInfo fileInfo = new FileInfo(filePath);

public void browse(){
  if(!checkExtension(fileInfo.getExtension()))
    return;
  String content = fileInfo.readContent();
  System.out.println(content);
}

public boolean checkExtension(String ext){
  if(ext.equals(extension))
    return true;
  return false;
}
```

Listing 4.2: Data Dependency Example

Listing 4.2 demonstrates an example on data dependency where three fields are declared at the beginning of the code fragment - *extension*, *filePath* and *fileInfo*. The example code snippet also contains a method named *browse* that uses *fileInfo* in its body. Since *fileInfo* depends on the field *filePath*, *browse* relies on both *fileInfo* and *filePath* for its execution. Again, *browse* depends on another field *extension* because it invokes a method named *checkExtension* which uses the field. In order to make the method *browse* executable, dependencies on these three fields should be identified and resolved.

4.2.3 Library Dependency

Libraries are developed to reuse the source code easily and make the development process faster. Usually, common and basic functionalities are exposed in the form of Application Program Interface (API) in the libraries. Developers use these APIs

with some adaptations and extensions to implement a particular feature. This type of usage indicates that a method may use one or more functions of a library to accomplish its intended task. To transform the method into a self-executable code fragment, all the libraries used by it should be resolved. Otherwise, it will get compilation error for not having the libraries in the development environment.

```
1 import java.io.File;
2 import java.io.FileWriter;
3 import java.io.IOException;
4
5 public class FileUtility {
6
7 public File createFile(String data, String filePath) throws
    IOException {
8   File file = new File(filePath);
9   FileWriter fileWriter = new FileWriter(file);
10  fileWriter.write(data);
11  fileWriter.close();
12  return file;
13 }
14 }
```

Listing 4.3: Library Dependency Example

An example is shown in Listing 4.3 where a method *createFile* uses three libraries to create file and write data in the file. In order to work with file in Java, *createFile* uses the library *java.io.File*. To write the content in a file, the method uses *java.io.FileWriter*. Another library *java.io.IOException* is used by the method to handle exceptional cases while creating and writing a file, for example, file not found, error in writing a file, file creation error, etc. To make the method *createFile* self-executable, all the libraries on which it is dependent should be detected and resolved.

4.2.4 Resolution of Method Call Dependency

In order to resolve method call dependency for a method (m), a ***call graph*** is generated by parsing the source code and identifying method invocation statements. Here ***call graph*** is defined as follows.

Definition 4. *A **call graph** is a directed graph G(E, V), where each vertex $v \in V$ denotes a unique method, and each edge $e(a, b) \in E$ denotes a calling relationship from a to b that is a invokes b.*

Next, the graph is traversed to find all the precedences of m and these are stored as the dependent methods.

```
1  public void f1(){
2    f2();
3  }
4  public void f2(){
5  }
6  public void foo1(){
7    foo2();
8    foo3();
9  }
10 public void foo2(){
11   foo4();
12 }
13 public void foo3(){
14 }
15 public void foo4(){
16 }
```

Listing 4.4: Call Graph Example

Figure 4.2 depicts the call graph of the example code snippet shown in Listing 4.4. There are six methods in the code fragment which are $f1$, $f2$, $foo1$, $foo2$, $foo3$ and $foo4$. As $f1$ invokes $f2$, an edge is drawn from node f1 to node f2 in the

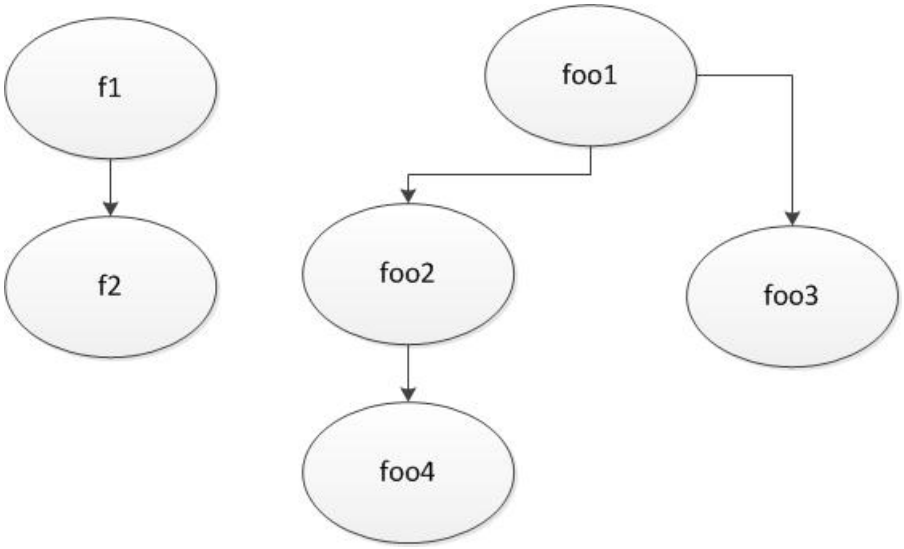

Figure 4.2: Call Graph of Listing 4.4

graph. Since $foo1$ calls $foo2$ and $foo3$ directly, two edges are found from $foo1$ to $foo2$ and $foo3$, respectively. Due to invoking $foo4$ indirectly by $foo1$ through $foo2$, no edge is found from $foo1$ to $foo4$. However, a path exists from $foo1$ to $foo4$ ($foo1 \rightarrow foo2 \rightarrow foo4$) in the graph to denote that $foo1$ is indirectly dependent to $foo4$.

In order to construct call graph from a given source file and find method call dependencies from the graph, Algorithm 1 is proposed. In the algorithm, the procedure $ConstructCallGraph$ takes a source code file as input for which call graph will be generated. The source file is parsed to obtain all the declared methods and store those into M. A map named *adj* is declared in Line 3 of Algorithm 1 to store calling relationship among the methods. For each method m in M, all the statements found in the method body are parsed to get method invocation statements as shown in Algorithm 1 from Line 5 - 10. Method name, argument type and return type are obtained from the invocation statement to match with the signature of other methods. If a matching is found for any of the declared methods, the respective method is stored in *adj* as an invoked method of m (Algorithm 1, Line 11-12). Next, the procedure returns *adj* as shown in Algorithm 1, Line 18. The second procedure named $FindInvokedMethods$ uses

Algorithm 1 Generating Call Graph And Finding Invoked Methods

Input: A source file (F) for which call graph will be generated

```
procedure ConstructCallGraph(F)
    parse F to find all the methods and store into M
    Map < Method, List < Method >> adj
    for each m ∈ M do
        for each st ∈ M.body.statements do
            if st is a method invocation statement then
                for each m′ ∈ M do
                    if m == m′ then
                        continue
                    end if
                    if st.name == m.name & st.argumentsTypes == m.parametersTypes & st.returnType == m.returnType then
                        adj[m].add(m′)
                    end if
                end for
            end if
        end for
    end for
    return adj
end procedure
procedure FindInvokedMethods(m, adj)
    Y = ∅
    Queue q = ∅
    q.push(m)
    while q ≠ ∅ do
        x = q.pop()
        Y = Y ∪ x
        for each x′ ∈ adj[x] do
            q.push(x′)
        end for
    end while
    return Y
end procedure
```

adj to find all the method invoked by m which is given as input. In the procedure a list named Y is declared to store the invoked methods. A queue Q is initialized with m to find all the methods that are accessible from m using *adj*. In each iteration of the *while* loop, the queue is popped to get the unvisited method (x) in the call graph (*adj*). Next, x is added to Y as an invoked method of m. All the methods called by x are obtained and added to Q using a *for* loop in Algorithm 1 Line 27-29. After finishing the iteration of the *while* loop, the procedure return Y as the list of methods on which m is dependent.

Complexity Analysis of Algorithm 1

Assume that,

m = number of methods

s = number of statements in a method

e = number of edges in the call graph

The complexity of the procedure $ConstructCallGraph$ is $O(m^2s)$. The procedure $FindInvokedMethods$ follows Breadth First Search (BFS) in the generated call graph to find the methods invoked by a given method. So, the complexity of the procedure is $O(m + e)$.

4.2.5 Resolution of Data Dependency

A method (m) may depend on one or more fields for its execution as stated earlier. So, the declaration statements of these fields need to be identified to make the method self-executable. In order to find the field on which m is dependent indirectly, a call graph is generated for m. The reason is that a field may be used by m indirectly through one of its invoked methods. Each statement found in m and its invoked methods are parsed to check whether it manipulates or uses any field declared outside the method body. Fields that are found in any of the statements, are stored as the dependent fields of m.

In order to resolve data dependency among the fields, a ***data dependence graph*** needs to be generated by parsing field declaration statements. Here, ***data dependence graph*** is defined as follows.

Definition 5. *A **data dependence graph** is a directed graph G(E,V) where each vertex $v \in V$ denotes a field and each edge $e \in E$ represents the dependency between two fields. If $a, b \in V$, an edge from a to b ($a \rightarrow b \in E$) denotes that a is dependent on b for its execution.*

```
1  int a;
2  a=10;
3  int b;
4  b=20;
5  int c=a+b;
6  int d=c*c+2*a;
7  int e=d+2*c-a;
```

Listing 4.5: Sample Code Snippet for Data Dependece Graph

A data dependence graph is shown in Figure 4.3 for the sample code snippet Listing 4.5. There are five fields in the code fragment which are a, b, c, d and e. Since field c depends on a and b as shown in Listing 4.5 line 5, two edges - $c \rightarrow a$ and $c \rightarrow b$ are seen in the figure. d relies on c and a for its initialization, so d is connected to a and c through the edges - $d \rightarrow a$ and $d \rightarrow c$. No outgoing edge is seen for a and b because these fields do not depend on any other fields for the initialization.

To construct data dependency graph from a given source file F, a procedure named $ConstructDataDependencyGraph$ is shown in Algorithm 2. In the procedure, all the statements in F are parsed to identify the fields declared in the source code. A map G is declared in Algorithm 2 line 4 to store which field is dependent on which other fields. The *for* loop iterates on the statements to detect field declaration or initialization statement (Algorithm 2 Line 5-12). If a declara-

Algorithm 2 Resolving Field Dependency

Input: A source file (F) for which data dependency graph will be generated

```
1: procedure ConstructDataDependencyGraph(F)
2:     statements = parse F to find all the statements in F
3:     fields = parse F to obtain all the fields
4:     Map < Field, List < Field >> G
5:     for each s ∈ statements do
6:         if s is field declaration or initialization statement then
7:             f = fields in s
8:             x = declared field in s
9:             y = f − x
10:            G[x] = G[X] ∪ y
11:        end if
12:    end for
13:    return G
14: end procedure
15: procedure FindUsedField(M, F, G)
16:    D = ∅
17:    P = ∅
18:    P = P ∪ M ∪ FindInvokedMethod(M)
19:    for each p ∈ P do
20:        for each s ∈ p.body.Statements do
21:            F = parse fields from s
22:            for each f ∈ F do
23:                D = D ∪ f ∪ Visit(f, G)
24:            end for
25:        end for
26:    end for
27:    return D
28: end procedure
29: procedure Visit(nd, G)
30:    Y = ∅
31:    Queue q = ∅
32:    q.push(nd)
33:    while q ≠ ∅ do
34:        x = q.pop()
35:        Y = Y ∪ x
36:        for each x′ ∈ G[x] do
37:            q.push(x′)
38:        end for
39:    end while
40:    return Y
41: end procedure
```

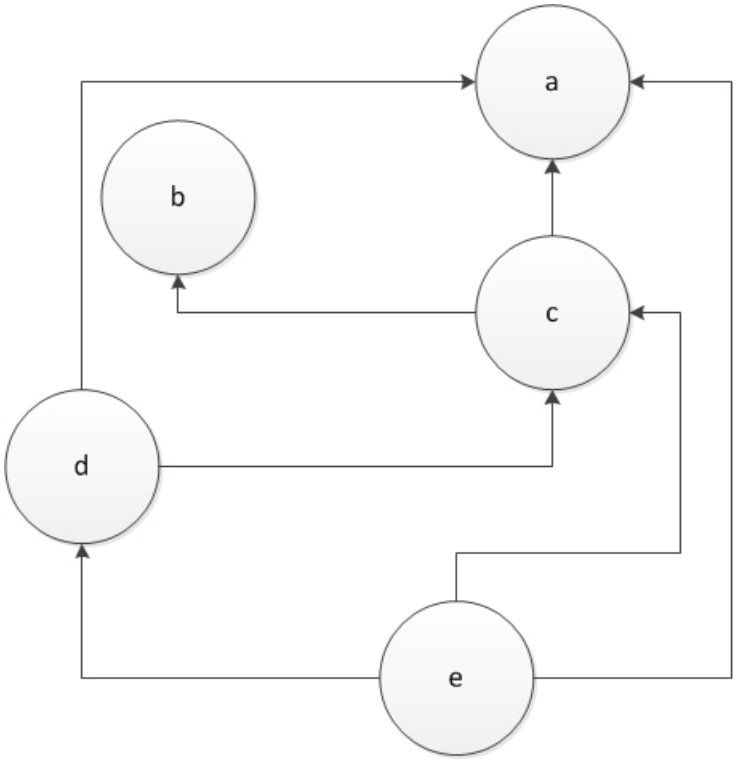

Figure 4.3: Data Dependence Graph of Listing 4.5

tion statement is found, it is parsed to identify declared field x and its dependent fields y (Algorithm 2 Line 7-9). Later, y is stored into G against the field x as shown in Algorithm 2 line 10. At last, the procedure return the generated data dependence graph G for F (Algorithm 2 Line 13).

Another procedure named $FindUsedField$ is declared in Algorithm 2 line 15 to identify all the fields on which a method m depends. In the procedure, a list D is declared to store the dependent fields (Algorithm 2 Line 16). Since m may use one or more fields indirectly through invoking other methods, a list P stores all the methods invoked by m (Algorithm 2 Line 18). For each method p in P, all the statements are parsed to check the occurrence of any field $f \in F$. If such an occurrence is found, a procedure named $Visit$ is called to find the fields on which f is dependent (Algorithm 2 Line 23). The procedure traverses the data dependence graph G to find all the nodes accessible from f (Algorithm 2 Line 30-40). Later, f and its dependent fields are added to D as shown in Algorithm 2 line 23. After

finishing the iteration on all the methods invoked by m, the procedure returns D as the final list of fields on which m relies (Algorithm 2 Line 27).

Complexity Analysis of Algorithm 2

Assume that,

m = number of methods

f = number of fields

s = number of statements in a source file

e = number of edges in the graph

In Algorithm 2, the procedure $ConstructDataDependencyGraph$ uses a single *for* loop that iterates on all the statements s. So, the complexity of the procedure is $O(s)$. Again, the complexity of the procedure $Visit$ is $O(f + e)$ as it follows BFS to find all the dependent fields of a given field. The procedure $FindUsedField$ invokes $FindInvokedMethod$ as shown in Line 18 Algorithm 2. The procedure uses a nested *for* loop and invokes another procedure $Visit$ (Algorithm 2 Line 19 - 26). So the overall complexity of $FindUsedField$ is as follows.

$$\text{Complexity of } FindInvokedMethod + \text{Complexity of nested for loop with } Visit \text{ invocation}$$

$$= m + e + m * s * f * (f + e)$$

$$= m + e + (m * s * f^2 + m * s * f * e)$$

$$= O(m * s * f^2 + m * s * f * e)$$

4.2.6 Resolution of Library Dependency

A method may use one or more external libraries to implement its intended feature. To make the method self-executable, these libraries need to be identified and

added to the development context. To find the libraries on which the method is dependent, Algorithm 3 is proposed.

Algorithm 3 Resolving Library Dependency

Input: A method (m) with its source file (F) which will be parsed to identify the libraries used by m

```
1:  procedure ResolveLibraryDependency(m, F)
2:      L = Libraries parsed from F
3:      CodeFragment cf;
4:      for each l ∈ L do
5:          cf.add(l)
6:      end for
7:      callgraph = ConstructCallGraph(F)
8:      M = FindInvokedMethod(m, callgraph)
9:      for each m′ ∈ M do
10:         cf.add(m′)
11:     end for
12:     dg = ConstructDataDependenceGraph(F)
13:     F = FindUsedField(m, F, dg)
14:     for each f ∈ F do
15:         st ← find initialization statement for f
16:         cf.add(st)
17:     end for
18:     x = ∅
19:     for each l ∈ L do
20:         cf.remove(l)
21:         if isCompilable(cf) == false then
22:             cf.add(l)
23:             x = x ∪ l
24:         end if
25:     end for
26:     return x
27: end procedure
```

In Algorithm 3, a procedure named *ResolveLibraryDependency* takes two inputs - a method m and its source file F, to identify the libraries used by m. A list L is declared to store all the fields by parsing the source file F. To store self-executable code snippet for m, a variable cf is declared in Algorithm 3 line 3. A *for* loop iterates on L to insert all the libraries into cf (Algorithm 3 Line 4 - 6). m may use a library indirectly through invoking another method. To identify the method invocation sequence in F, a call graph (*callgraph*) is generated by calling

the procedure *ConstructCallGraph* (defined in Algorithm 1). To obtain all the methods called by m, the procedure $FindInvokedMethod$ presented in Algorithm 1, is invoked with the arguments - *callgraph* and m. A *for* loop is defined in Algorithm 3 line 14 to insert all the invoked methods into cf. Two procedures - $ConstructDataDependenceGraph$ and $FindUsedField$ (presented in Algorithm 2) are invoked to find all the fields on which m is dependent (Algorithm 3 Line 12-13). The initialization statements for the fields are identified and added to cf as shown in Algorithm 3 line 14-16. A list x is declared to store all the libraries on which m is dependent (Algorithm 3 Line 18). The *for* loop iterates on the L to find the dependent libraries. For each library $l \in L$, it is checked whether cf is compilable if l is removed from cf. If cf is not compilable, l is required for running n and it is added to x (Algorithm 3 Line 19 - 25). At last, the procedure returns x that contains all the libraries required for the execution of m.

Complexity Analysis of Algorithm 3

Assume that,

l = number of libraries

m = number of methods

f = number of fields

s = number of statements in a source file

e = number of edges in the graph

The complexity of the procedure *ResolveLibraryDependency* is as follows.

$$\text{for loop in the 4th line} + \text{invocation of } ConstructCallGraph$$
$$+\text{invocation of } FindInvokedMethod + \text{for loop in the 9th line}$$
$$+\text{invocation of } ConstructDataDependenceGraph$$
$$+\text{invocation of } FindUsedField + \text{for loop in the 14th line}$$
$$+\text{for loop in the 19th line}$$

$$= l + m^2 * s + (m + e) + m + s + (m * s * f^2 + m * s * f * e) + f + l$$

$$= O(m * s * f^2 + m * s * f * e)$$

4.3 Method Signature Regeneration

Although the signature of a method expresses the input and output types, this is not sufficient enough to convert into self-executable function for three scenarios. These are given as follows.

1. A method may have return type *void* but it may manipulate one or more variables that are declared outside the body of the method.

2. A method may not have any parameter (that is, *void*) but may use variables that are defined outside the body of the method.

3. The signature of a method may explicitly state the input and output types but some variables may be used or manipulated by it and these are declared outside the method body.

Considering all of these scenarios, the technique generates data dependency graph to redefine the signature and convert into reusable method. Each node in the graph denotes the variable and an edge from a to b ($a \rightarrow b$) denotes variable a depends on variable b. After constructing the graph, nodes that have in-degree

zero and variables denoted by these nodes are declared outside the method body, are considered as input parameters. Again, nodes that have out-degree zero are considered as output variables of the method. If multiple output variables are found, a complex data type is created where each field of the type denotes each node. The reason is that a method return type can be a single data type - either primitive or complex data type.

To generate the parameters of the method, the variables found in the nodes containing in-degree zero are used. If a single node is found with zero out-degree, the type of the variable denoted by the node is used as return type of the method. Otherwise, generated composite data type as discussed earlier is used. The signature of the method is redefined by combining the return type, method name and parameters. It is possible to have one or more variables that are declared outside the method body. In the data dependency graph, nodes representing these variables may have at least one in-degree and one out-degree. In this case, the source code is parsed and the declaration statements of the variables are checked to determine the types of the variables. Using this information, it adds declaration statement for each of the variables at the beginning of the function body. Thus, the technique redefines the signature of the self-executable method.

4.4 Input Dataset Generation

In order to determine the behavior of a method, SBMF executes the method against a set of input values and matches the generated output set. A method exposes different behaviors due to having multiple conditional branches. Output values may vary from one branch to another in the same method. An example code snippet is shown in Listing 4.6 to explain how a method differently behaves.

```
1 int calculate(int num1, int num2, char op){
2   int ans;
3   if(op=='+')
```

```
4       ans=num1+num2;
5     else if(op=='-')
6       ans=num1-num2;
7     else if(op=='*')
8       ans=num1*num2;
9     else
10      ans=num1%num2;
11    return ans;
12 }
```

Listing 4.6: Sample Code Snippet for Demonstrating Different Behaviors

In the code snippet, the method *calculate* takes three parameters which are $num1$, $num2$ and *op*. *calculate* performs four types of operations based on the value of *op* which are given below.

1. If *op* is '+', the method will add two numbers - $num1$ and $num2$
2. If the value *op* is '-', the method will subtract $num2$ from $num1$
3. If '*' is provided to *op*, $num1$ will be multiplied by $num2$
4. For any other value of *op*, the method will return the remainder of $num1$ and $num2$

Due to having four conditional branches, the method exposes any of the four types of behavior in each execution. SBMF needs to identify all possible behaviors to check feature-wise similarity among a set of methods. In order to determine all possible tasks performed by a method, input data set should cover all the branches found in the method body. In this step, a ***Control Flow Graph (CFG)*** is generated to identify all the branches in a method body. Here CFG is defined as follows.

Definition 6. *A* ***Control Flow Graph*** *G(V,E) is a directed graph representing the flow of control in the program. Each node* $v \in V$ *denotes sequential blocks of*

program commands or statements. Each edge $e \in E$ represents the dependency relationship between two nodes. Multiple edges from a node indicate conditional statements (loops, if statements, switches, etc.).

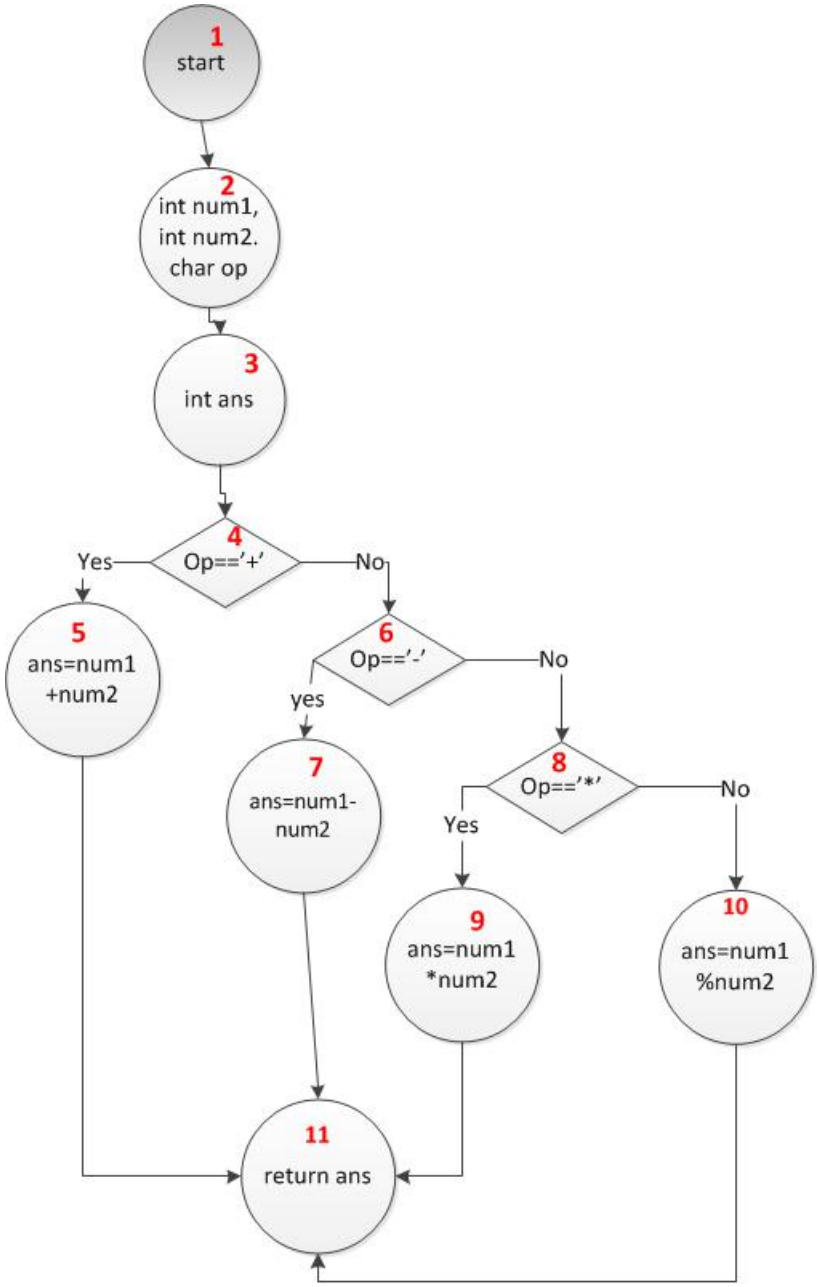

Figure 4.4: Control Flow Graph for Listing 4.6

A CFG for Listing 4.6 is shown in Figure 4.4. As there are three conditional statements, the figure contains three diamond shapes to represent the conditional branches. According to the figure, there are four possible execution paths from

the node *start* to *return ans* and these are -

- Path-1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 11$
- Path-2: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11$
- Path-3: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 11$
- Path-4: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11$

Each path represents a specific behavior of the method, for example, Path-1 denotes the addition operation on $num1$ and $num2$. In order to identify all possible behaviors of a method, SBMF finds all paths from the initial statement to the return statement of the method. For each path p, it finds the parameters that cause the method to execute p. Constraints are also extracted from the conditional statements of p to select data for the parameters. For example, in order to execute the Path-1, a constraint is that the argument for *op* must be '+'. After obtaining all the constraints for p, Boundary Value Analysis (BVA) [109, 110] is carried on all the parameters to generate input set. According to BVA, the maximum and minimum possible values are determined from the data type of a parameter. Next, a random value is generated within the boundary of the parameter. All these values are generated without violating the constraints of p. The final input dataset for p is created by generating all possible combinations of parameter values. For instance, the upper limit of *int* data type is 2147483647 and the lower limit is -2147483647. Assume that the generated random value is 0. There are two *int* parameters in *calculate* ($num1$ and $num2$) and a *char* parameter *op*, which value must be '+' to execute Path-1. So, the input data set for $num1$ is {-2147483647, 0, 2147483647}, for $num2$ is {-2147483647, 0, 2147483647}, and for *op* is {'+'}. After taking Cartesian product of these sets, the final input data set for Path-1 will be { (-2147483647, -2147483647, '+'), (-2147483647, 0, '+'), (-2147483647, 2147483647, '+'), (0,-2147483647, '+'), (0, 0,

'+'), (0, 2147483647, '+') (2147483647, -2147483647, '+'), (2147483647, 0, '+'), (2147483647, 2147483647, '+') }

4.5 Clustering Similar Methods

To improve the code search, it is required to check the similarity among methods found in the code base. Two or more methods may perform the same task in different ways. So, feature-wise similar methods need to be detected to retrieve the similar methods all together.

For a given set of self-executable methods M, similarity is checked by running each method $m \in M$ and checking the output. A set of input values I is generated for $m \in M$ following Step-3 (Input Dataset Generation) and corresponding set of output values O is obtained through executing m. For each $m' \in M$ and $m \neq m'$, m' is said to be functionally similar if its output set O' for I is the same as O. Accordingly, methods are clustered based on the feature-wise similarities where each cluster contains the methods that perform the same task. That is, a cluster $C \subset M$ and $\forall x, y : C \cdot\ x$ and y are functionally similar. The approach to cluster the methods based on the feature-wise similarity is presented in Algorithm 4.

In Algorithm 4, the procedure named $ClusterSimilarMethods$ takes a list of reusable methods M as input which is constructed following the previous step. A variable C is declared to store different clusters of similar methods where each cluster contains the methods that perform the same functionality (Algorithm 4 Line 2). A *for* loop is declared that iterates on M to construct cluster of similar methods. The procedure $IsInAnyCluster$ is invoked to check whether each method m (belongs to M) is previously added to any cluster or not (Algorithm 4 Lines 4-5). If m does not belong to any cluster, a variable cl is declared to contain all the methods similar to m. A set of input data is generated based on the type of parameters found in the signature of m and corresponding output is generated by

executing m (Algorithm 4 Lines 9-10). Here *inputset* and *outputset* determine the intent of m. Another *for* loop is declared to identify the other methods that are similar to m. In each iteration, the signature of each method m' (in M) is matched with the signature of m to check whether the input dataset can be fed into the method and return type is identical to m (Algorithm 4 Line 15). If the signatures of both the methods are identical, the method m' is executed for *inputset* and generated output is stored to *outputset*$'$. If *outputset* and *outputset*$'$ are found the same, m' is considered similar to m as both methods produce the same output for the same input data set (Algorithm 4 Lines 17-19). m' is then added to cl to store all the methods similar to m. At last, cl is inserted to the list of all identified clusters (C).

There may have different techniques to implement the same feature, for example, sorting can be implemented following bubble sort, merge sort, etc. All the sorting techniques will be in the same cluster as these are functionally similar but implementation-wise different. It would not be good to retrieve codes implementing bubble sort when a user looks for merge sort. Each cluster C is decomposed into a set of sub-clusters R based on time and memory space complexities. That is, $R \subseteq C$ and $\forall r : R \cdot$ ($\forall p, q : r \cdot O(p) = O(q)$ and p, q are functionally similar).

Complexity Analysis of Algorithm 4

Assume that,

m = number of methods

c = number of cluster

The procedure $IsInAnyCluster$ contains only a *for* loop that iterates on each cluster in c. For each cluster the procedure checks whether the given method resides in the cluster and it takes $O(m)$ for each checking. In the worst case, the size of the cluster equals to the number of methods when each cluster contains a single method (that is $c = m$). So, the complexity of the procedure is $O(m^2)$.

Again, the complexity of the procedure $ClusterSimilarMethods$ uses a for loop that iterates m times and $IsInAnyCluster$ is invoked in each iteration. So, the complexity of $ClusterSimilarMethods$ is $O(m^3)$

4.6 Proper Term Selection

In order to retrieve more relevant methods, it is required to identify proper terms for each method before indexing. When two or more methods have different names or signatures, but these implement the same functionality, these methods should be indexed under common appropriate terms. As a result, all these methods will be obtained against the user query. So, after getting all the clusters from the previous step, representative terms needs to be identified for each cluster. For a cluster, terms are obtained from the methods found in the cluster through extracting, tokenizing and stemming keywords found in the methods. Terms that are found in most of the methods are considered as final representative terms for each of these methods.

In Algorithm 5, a procedure named $SelectProperTerm$ is declared to find relevant terms from a given cluster of methods C. To store the frequency of methods for each term extracted from $m \in C$, a map $frequency$ is declared. A set T is declared to store all the unique terms found in c (Algorithm 5 Line 3). A *for* loop is defined in Algorithm 5 line 4 - 18 which iterates on each method $m \in c$ to obtain terms and calculate frequency. In each iteration, keywords found in m are tokenized and stemmed to generate terms. To have a list of candidate proper terms, these terms are added to T. A set *used* is declared to keep track of the unique terms in m. The inner *for* loop iterates on the generated terms to calculate frequency of each unique term (Algorithm 5 Line 9-17). For each term t, it is checked whether t is already added to $frequency$ or not (Algorithm 5 Line 10). If t is a new term, t is added to the map $frequency$ with initial value zero.

Algorithm 4 Cluster Similar Methods

Input: A list of methods (M) for which search index will be constructed

```
procedure ClusterSimilarMethods(M)
    C = ∅;
    for each m ∈ M do
        if IsInAnyCluster(m, C) == true then
            continue
        end if
        cl = ∅
        cl.add(m)
        inputset = generate a set of input data randomly for m
        outputset = execute m and generate corresponding output for inputset
        for each m′ ∈ M do
            if IsInAnyCluster(m′, C) == true then
                continue
            end if
            if m′.paramtersTypes == m.parametersTypes & m′.returnType = m.returnType then
                outputset′ = execute m′ and generate corresponding output for inputset
                if outputset == outputset′ then
                    cl.add(m′)
                end if
            end if
        end for
        C.add(cl)
    end for
end procedure
procedure IsInAnyCluster(m, C)
    found = false
    for c ∈ C do
        if m ∈ c then
            found = true
            break;
        end if
    end for
    return found
end procedure
```

Algorithm 5 Proper Term Selection

Input: A cluster (c) for which proper index term will be generated

```
1: procedure SelectProperTerm(c)
2:     Map < String, int > frequency
3:     T ← ∅
4:     for each m ∈ c do
5:         tokens ← tokenize(m.name)
6:         terms ← stem(tokens)
7:         T = T ∪ terms
8:         used ← ∅
9:         for each t ∈ terms do
10:            if t ∉ frequency.keys then
11:                frequency[t] = 0;
12:            end if
13:            if t ∉ used then
14:                used = used ∪ t
15:                frequency[t] + +
16:            end if
17:        end for
18:    end for
19:    relevantTerms ← ∅
20:    for t ∈ T do
21:        X = c.size()
22:        Y = frequency[t]
23:        if X ≥ Y/2 then
24:            relevantTerms = relevantTerms ∪ t
25:        end if
26:    end for
27:    return relevantTerms
28: end procedure
```

Another conditional statement is shown in line 13 to check whether t is a new term for m or not. If t is found as a new term for m, the frequency of t is incremented by one as shown in Algorithm 5 line 15. A set named $relevantTerms$ is declared to hold all the relevant terms for indexing the cluster c (Algorithm 5 Line 19). A *for* loop is used to select proper term from T. In each iteration, the number of methods in c is obtained and stored in X. In addition, a variable Y holds the number of methods that contains the term $t \in T$. The conditional statement shown in Algorithm 5 Line 23 checks whether t appears in most of the methods or not. If frequency of t is greater than or equal to half of the cluster size, t is added to $relevantTerms$ (Algorithm 5 Line 24). After finishing iteration on T, the procedure returns $relevantTerms$ that will be used to index the cluster in the next step.

Complexity Analysis of Algorithm 5

Assume that,

m = number of methods

t = number of terms

The complexity of the procedure $SelectProperTerms$ in Algorithm 5 is shown as follows.

$$\text{for loop in the 4th line} * \text{for loop in the 9th line} + \text{for loop in the 20th line}$$

$$= m * t + t$$

$$= O(m * t)$$

4.7 Index Construction

After generating appropriate terms for each cluster of methods, an index is built for searching desired methods. A posting list is created to construct index, which maps terms with corresponding methods. When a user query is submitted, query terms are matched with the index terms in the posting list to retrieve relevant methods. The mechanism to construct the index is shown in Algorithm 6.

A procedure named *ConstructIndex* is presented in Algorithm 6 to build index of methods obtained from the previous steps. To construct the index, an empty posting list is declared, which maps each term to corresponding methods (Algorithm 6 Lines 2). A nested *for* loop is defined, where the outer loop iterates on a list of methods (M) given as input to the procedure (Algorithm 6 Lines 3-4). The inner loop iterates to get all the terms of each method in M. In addition, each term is checked whether the posting list contains it or not to add a new term in the list (Algorithm 6 Lines 5-7). Next, the method is added to the posting list against the term so that, when a query term will match with the term, the method will be retrieved (Algorithm 6 Lines 8). After adding all the methods, the list is returned by the procedure (Algorithm 6 Lines 11).

Algorithm 6 Index Construction

Input: A list of methods (M) containing signature, body and terms of each method

```
1:  procedure ConstructIndex(M)
2:      Map < String, List < Method >> postingList
3:      for each m ∈ M do
4:          for each t ∈ m.terms do
5:              if !postingList.keys.contains(t) then
6:                  postingList.keys.add(t)
7:              end if
8:              postingList[t].add(m)
9:          end for
10:     end for
11:     return postingList
12: end procedure
```

Complexity Analysis of Algorithm 6

The procedure $ConstructIndex$ in Algorithm 6 contains a nested *for* loop where the outer loop iterates on a list of methods and the inner loop iterates on a list of terms. If m is the number of methods and t is the number of terms, the complexity of the procedure is $O(m * t)$.

4.8 Query Reformulation

In order to expand query with context and topic specific keywords, SBMF adopts user query logs containing the query terms and the code fragments or software artifacts that are clicked. Each user query in the query logs is converted into document vector. A term-term matrix is constructed where semantically and contextually similar terms are stored in the same row of the matrix. Here, the similarity score is calculated by applying Jaccard Similarity [111] formula on term co-occurrences in the same query. At last, for a given user query, top k relevant terms are obtained from the matrix to expand each term. The expanded query is then submitted to the search engine to retrieve relevant code snippets. All these tasks are discussed in the following subsections.

4.8.1 Data Preprocessing

In this step, user query logs are processed to construct a collection of conceptually similar words that act as data source for query expansion. SBMF obtains query logs from historical repository and converts each query into document (that is, collection of terms). Each keyword in the query is checked whether it appears in the retrieved codes that are clicked or downloaded against the query. Keywords that appears at least once are considered relevant, and document is constructed by tokenizing and stemming these keywords.

In Algorithm 7, a complex data type named $QDoc$ is defined which has two

fields (*documents* and *query*) to represent each query obtained from the query log repository. The field *query* stores query keywords, and *documents* holds source codes that are clicked or downloaded against the corresponding query. The procedure *ConvertQueryToDoc* takes a list of *QDoc* and creates a collection of documents from these. A nested *for* loop is employed where the outer *for* loop iterates on *qDocs* to convert each query into document (Algorithm 7 Line 3-18). For each item in *qDocs*, a list of *String* (*docKeywords*) is defined to store terms from the retrieved codes that are clicked or downloaded. The first inner *for* loop iterates on these codes and invokes another procedure named *Process* to get terms from each source code. The procedure *Process* takes source code as input and removes punctuations from the code (Algorithm 7 Line 21-22). After that, it generates tokens and converts each token into term by employing stemming operation. At last, the stop words are removed and a list of all valid terms are returned (Algorithm 7 Line 31). After receiving terms for each *qDocs*, the list *docKeywords* stores these for pruning irrelevant query terms (Algorithm 7 Line 7). Another list named *queryTerms* is used to store all the query terms returned by the procedure *Process* for each query *qDoc.query*. A *for* loop is defined which iterates on *queryTerms* to find irrelevant terms that are not available in *docKeywords* (Algorithm 7 Line 12). To generate document for the query *qDoc.query*, all the valid terms are stored in a list named *document* (Algorithm 7 Line 14). The document is added to the variable *documents* to create collection of documents which is returned by the procedure *convertToDoc*.

Complexity Analysis of Algorithm 7

Assume that,

q = number of query docs

d = number of clicked docs

t = number of tokens

Algorithm 7 Data Processing

```
procedure ConvertQueryToDocument(qDocs)
    List < List < String >> docs
    for each qDoc ∈ qDocs do
        List < String > docKeywords
        for each doc ∈ qDoc.documents do
            keywords = Process(doc)
            docKeywords.addAll(keywords)
        end for
        List < String > queryTerms
        queryTerms = Process(qDoc.query)
        List < String > document
        for each term ∈ queryTerms do
            if term ∈ docKeywords then
                document.add(term)
            end if
        end for
        docs.add(document)
    end for
    return docs
end procedure
procedure Process(List < String > doc)
    doc = removePunctuation(doc)
    List < String > processedList, tokens
    tokens = tokenize(doc)
    for each token ∈ tokens do
        word = convertToRootForm(token)
        if word ∉ stopWords then
            processedList.add(word)
        end if
    end for
    return processedList
end procedure
```

The procedure *Process* in Algorithm 7 iterates on a list of tokens as shown in Line 25. So, the complexity of the procedure is $O(t)$. Algorithm 7 also contains another procedure named *ConvertQueryToDocument* and the complexity of the procedure is shown below.

$$\text{for loop in the 3rd line} * (\text{for loop in the 5th line} * \text{invocation of } Process$$
$$+\text{for loop in the 12th line})$$
$$= q * (d * t + t)$$
$$= O(qdt)$$

4.8.2 Term-Term Matrix Construction

In order to find conceptually similar terms, a Term-Term matrix is constructed from the document collection that is created in the previous step. Each row in the matrix denotes a posting list for a particular term where the list contains all the related terms. Terms that occur in the same query are considered conceptually related to each other. So, these are ranked according to their co-occurrences in the query logs. To expand a particular query term, top ranked terms related to the query term are selected from the list.

The procedure *ConstructMatrix* in Algorithm 8 takes a list of documents *docs* obtained from the previous step and constructs collections of conceptually similar terms. A map named *matrix* is created to store these terms and an empty list *terms* is declared to save all the terms found in *docs*. A nested *for* loop is defined where the outer loop iterates on every document (*doc*) in *docs* and inner loop inserts terms into *terms* found in the document *doc* (Algorithm 8 Line 4-10). Another nested *for* loop is used to calculate similarity score between every pair

of terms in *terms* (Algorithm 8 Line 11-31). For each term $t1$ in *terms*, a list $cTerms$ is declared to contain all the terms conceptually related to $t1$. $CTerm$ is a composite data type which stores term and similarity score by using *term* and *score* attributes respectively. In the inner *for* loop, the variables $nt1$ and $nt2$ are initialized with the number of documents that contain $t1$ and the number of documents that contain $t2$ respectively. Another variable $nt1Andt2$ stores the number of documents that contain both $t1$ and $t2$. Next, the number of documents having term $t1$ or $t2$ is calculated and stored in $nt1Ort2$. Later, Jaccard similarity score is calculated by taking ratio between $nt1Andt2$ and $nt1Ort2$ (Algorithm 8 Line 21). A variable $cTerm$ of type $CTerm$ is initialized with term $t2$ and the calculated score. After adding all the terms with respective similarity score, $cTerms$ is sorted by the score in descending order so that higher scored terms can be retrieved easily (Algorithm 8 Line 27). The list is inserted into the map *matrix* against the term $t1$ (Algorithm 8 Line 28-30). As a result, for a given term, its similar terms can be quickly retrieved from the *matrix*.

Complexity Analysis of Algorithm 8

Assume that,

d = number of documents

t = number of terms

The complexity of the procedure $ConstructMatrix$ in Algorithm 8 is as follows.

$$\text{for loop in the 4th line} * \text{for loop in the 5th line} + \text{for loop in the 11th line}$$
$$*(\text{for loop in the 13th line} + \text{for loop in the 28th line})$$
$$= d * t + t * (t + t)$$
$$= d * t + 2 * t^2$$

Algorithm 8 Term-Term Matrix Construction

```
1: procedure ConstructMatrix(docs)
2:     Map < String, List < String >> matrix
3:     List < String > terms
4:     for each doc ∈ docs do
5:         for each term ∈ doc do
6:             if term ∉ terms then
7:                 terms.add(term)
8:             end if
9:         end for
10:    end for
11:    for each t1 ∈ terms do
12:        List < CandidateTerm > cTerms
13:        for each t2 ∈ terms do
14:            if t1 == t2 then
15:                continue
16:            end if
17:            nt1 = get #Docs Containing Term t1
18:            nt2 = get #Docs Containing Term t2
19:            nt1Andt2 = get #Docs Containing t1 & t2
20:            nt1Ort2 = nt1 + nt2 − nt1Andnt2
21:            score = nt1Andt2/nt1Ort2
22:            CTerm cTerm = new CTerm()
23:            cTerm.term = t2
24:            cTerm.score = score
25:            cTerms.add(cTerm)
26:        end for
27:        cTerms = sort cTerms by score in descending order
28:        for each ct ∈ cTerms do
29:            matrix[t1].add(ct.term)
30:        end for
31:    end for
32:    return matrix
33: end procedure
```

Usually, a document contains at least one term that is $d \leq t$. So,

$$(d * t + 2 * t^2) \leq (t * t + 2 * t^2)$$

$$= 3 * t^2$$

$$= O(t^2)$$

4.8.3 Query Expansion

Usually, user query contains few keywords to express user needs. If these keywords are ambiguous, search engines may fail to retrieve many relevant code fragments or fetch irrelevant code snippets. To alleviate this ambiguity and represent information-needs more clearly, more topic specific keywords need to be added to the query. In this step, top scored terms are obtained for each query keyword from the term collection which is constructed in the previous step. Next, user query is reformulated by adding these terms and finally submitted to the search engine as expanded query.

Algorithm 9 Query Expansion

```
procedure ExpandQuery(query, matrix, k)
    String expandedQuery = ""
    for each term ∈ query do
        expandedQuery+ = "(" + term
        for each t ∈ matrix[term].top(k) do
            expandedQuery+ = "OR" + t
        end for
        if term ≠ query.lastElement() then
            expandedQuery+ = "AND"
        end if
        expandedQuery+ = ")"
    end for
    return expandedQuery
end procedure
```

In Algorithm 9, the procedure *ExpandQuery* has three parameters in its signa-

ture. The parameter *query* denotes the actual query provided by the user, *matrix* is a collection of conceptually similar terms produced by Algorithm 8, and k is the number of terms that will be retrieved to expand each query term. An empty string *expandedQuery* is declared to store the expanded version of the original query. A nested *for* loop is defined where the outer loop iterates on *query* to extract term from the query. For each term, top k conceptually similar terms are obtained from *matrix*. The inner *for* loop appends these to *expandedQuery* using boolean OR operation to keep the semantic information of *query* intact. After expanding all terms in *query*, the resultant expanded query *expandedQuery* is returned by the procedure.

Complexity Analysis of Algorithm 9

The procedure *ExpandQuery* in Algorithm 9 contains a nested *for* where the outer *for* loop iterates on a list of terms and the inner *for* loop iterates on a list of expanded terms. So, the complexity of the procedure is $O(t^2)$ where t is the number of terms.

4.9 Summary

In this chapter, a technique named Similarity Based Method Finder (SBMF) is proposed to retrieve more relevant methods from a code base. The technique first parses all the methods from the source code to construct a repository of methods. It generates data dependency graph for each method and converts the method into self-executable method (i.e., program slice) through resolving data dependency, and redefining parameters and return type. Later, all the methods are clustered into a number of clusters where methods in the same cluster perform the same task. To detect feature-wise similarity among a set of methods signatures (i.e., parameters and return types) of these methods are checked. Methods having

the same signature are then executed against a set of input values. Among these methods, those which produce the same output are considered as feature-wise similar and a cluster is constructed to store these methods. To identify proper terms for a cluster, keywords are obtained from the methods in the cluster and method frequency is calculated for each term. Such terms are considered as representative terms if these are found in most of the methods of the cluster. All the methods of the cluster are then indexed against the terms so that these are retrieved alltogether if a query term matches one of these methods. At last, user query is expanded by adding conceptually similar words of each query term to increase the matching probability between the query terms and index terms.

Chapter 5

Mathematical Analysis of SBMF

Existing code search engines index functionally similar codes against different terms. As a result, the engines cannot retrieve more relevant code fragments from a code base. Usually, the engines employ IR centric approaches to index the codes. Keywords in the source codes are used as index terms in these approaches. Many irrelevant codes are retrieved when these keywords are not relevant to the implemented features of the code snippets. This is termed here *Method Name Inconsistency*. Having ambiguous or irrelevant terms in the user query causes the engines to retrieve many irrelevant codes and it is commonly known as *Vocabulary Mismatch Problem*. Index terms may have hierarchical relationships among the code fragments. In order to retrieve more relevant codes, the relationships are required to be resolved. These problems reduce the effectiveness of the code search engines as many relevant codes may not be retrieved. In this chapter, the problems are mathematically modeled to show that these reduce the effectiveness in code search. To solve these problems, a technique named Similarity Based Method Finder (SBMF) has been proposed in the previous chapter. A mathematical analysis is given in this chapter to justify the solvability of SBMF. Here, SBMF is formalized to establish its relationship with the problems. Theoretical analysis is presented to justify that the technique increases the effectiveness in code search.

5.1 Definitions of Used Set Notations

Before getting into the mathematical models of the aforementioned problems, some terminologies are needed for a better understanding. Thus the used set notations in this thesis are provided in Table 5.1. Besides, the meanings of the notations are also presented in the table.

Table 5.1: Used Set Notations

Notation	**Meaning**
$\in$	Belongs To
$\notin$	Not Belong To
$\subset$	Proper Subset
$\subseteq$	Subset
$\Rightarrow$	Implication
$\longleftrightarrow$	Relation
$\mapsto$	Maplet (used to define "maps to")
$\cap$	Intersection
$\cup$	Union
$\wedge$	Conjunction
$\vee$	Disjunction
ran	Range of a Relation
$\forall$	For All (Universal Quantifier)
$\exists$	There Exists (Existential Quantifier)
$\bullet$	Predicate Separator
$\mathcal{P}$	Power Set
$x : A$	A variable x of type A
$\varphi x : a \mid p \bullet q$	φ is the quantifier, x is the bound variable, a is the range of x, p is the constraint, and q is the predicate.

5.2 Method Name Inconsistency

A method name expresses the implemented task or feature of the method. If the name is not consistent with the feature, it is termed as *Method Name Inconsistency*. Due to having irrelevant keywords in the method name, existing code search engines index the method against inappropriate terms. The engines cannot retrieve the code snippet against a valid user query where terms in the query represent the implemented feature properly. An example code snippet is shown Listing 5.1 which contains a method *S*. The method takes an array of integers as input and sorts in ascending order. It implements the algorithm *bubble sort* in its body. The method name 'S' is not consistent with the implemented feature. It would be better to have the name "sort" or "bubbleSort". When a developer looks for an example code of *bubble sort*, he may use the query "bubble", "sort" or "bubble sort". These query terms are relevant to the search intent but the method 'S' cannot be retrieved. The reason is that existing code search engines use 'S' to index the method and 'S' is not relevant to any of the query terms.

```
1  int[] S(int[] a){
2    for(int i=0;i<a.length;i++){
3      for(int j=0;j<a.length-1;j++){
4        if(a[j]>a[j+1]){
5          int temp=a[j];
6          a[j]=a[j+1];
7          a[j+1]=temp;
8        }
9      }
10   }
11   return a;
12 }
```

Listing 5.1: Example of Method Name Inconsistency

It is said that current development practice is to reuse and remix existing codes.

A large amount of open source code is available today which are reused by the developers to implement a given feature. Usually, developers put more focus on correct implementation of feature rather than naming convention in source code. Many code fragments are found in the open source repositories that have efficient implementation of different features. However, irrelevant keywords are found in these code snippets for not following naming convention properly. Code search engines fail to retrieve these snippets due to *Method Name Inconsistency*.

5.3 Mathematical Model of Method Name Inconsistency

Assume that a given code base contains a set of methods $M = \{m_1, m_2, m_3, ..., m_n\}$ and a set of functionalities $F = \{f_1, f_2, f_3, ..., f_n\}$. Each functionality $f \in F$ is implemented in one or more methods $m \in M$. The relationship from F to M can be denoted by $X = F \longleftrightarrow M$. Suppose, $T = \{t_1, t_2, t_3, ..., t_n\}$ is a set of terms which is used to construct the index. The relation $Y = F \longleftrightarrow T$ shows the relevant terms for a feature $f \in F$ and $Z = M \longleftrightarrow T$ denotes the terms found for each method $m \in M$. For a method $m' \in M$, the set of functionalities p implemented by m' can be obtained using the following set operation.

$$p = \{m : M; f : F \mid f \mapsto m \in X \wedge m = m' \bullet f\}$$

The set of relevant terms q for p can be derived from $F \longleftrightarrow T$ as follows.

$$q = \{t : T; f : F \mid f \mapsto t \in Y \wedge f \in p \bullet t\}$$

If r is a set of terms found in m', it can be calculated from $M \longleftrightarrow T$ as follows.

$$r = \{t : T; m : M \mid m \mapsto t \in Z \wedge m = m' \bullet t\}$$

According to the definition of *Method Name Inconsistency*. m' will be said to have the problem if $q \cap r = \emptyset$. That is, m' does not have any relevant term that represents its implemented functionalities.

In order to solve the problem, it is required to identify functionally similar methods from the code base. A set of methods $A \subseteq M$ is said to be functionally similar to each other if the methods implement the similar features.

To conduct feature-wise similarity analysis, it is required to find all possible functionalities exposed by a method. Different behaviors are shown by a method when it contains one or more conditional or branching statements. The reason is that different execution paths are followed based on the input values provided to the method. The execution behavior of a method can be modeled to a mathematical piece-wise function where each operation is defined by an execution path and constraints on the variables are defined by the conditional branches.

```
1  double convert(double celcius, string toScaleName){
2    if(toScaleName.equals("fahreinheit")){
3      return 1.8*celcius+32;
4    }
5    else if(toScaleName.equals("rankine")){
6      return (273.15+celcius)*1.8;
7    }
8    else if(toScaleName.equals("kelvin")){
9      return 273.15+celcius;
10   }
11   else{
12     return celcius;
13   }}
```

Listing 5.2: Sample Code to Demonstrate Different Execution Paths

An example code snippet is presented in Listing 5.2 which contains a method *convert*. The method takes the temperature value in celcius and converts it into fahreinheit, rankine or kelvin. The mathematical piece-wise function of the

method is shown below.

$$f(celcius, toScaleName) = \begin{cases} 1.8 * celcius + 32 & \text{if } toScaleName \in \{'fahreinheit'\} \\ (273.15 + celcius) * 1.8 & \text{if } toScaleName \in \{'rankine'\} \\ 273.15 + celcius & \text{if } toScaleName \in \{'kelvin'\} \\ celcius & \text{otherwise} \end{cases}$$

If the value of $toScaleName$ is 'fahreinheit', it converts the value of $celcius$ into fahreinheit scale as shown in the first case of the mathematical function. The value of $celcius$ is converted into rankine if $toScaleName$ is 'rankine' as shown in the second case of the function. If $toScaleName \in$ 'kelvin', the function will calculate the value in kelvin scale using the equation $273.15 + celcius$. This operation is shown in the third case of the function. It can be clearly seen that a method can be represented by a mathematical function. The function can be constructed by identifying all the branches and branching conditions.

SBMF finds all possible behaviors by generating an input set that covers all the branches in the method. Step-3 of SBMF (Input Dataset Generation) is responsible for the generation of the input set following Lemma 1.

Lemma 1. *Input set covering all branches exhibits all possible behaviors of a method*

Proof. Assume that a method m has a set of statements S and a set of branching statements B. Usually, a subset of statements are executed when the respective branching condition is true. It can be defined by a relation from branching conditions to statements that is $R = B \longleftrightarrow \mathcal{P}(S)$. An input set $I = \{i_1, i_2, i_3, ..., i_n\}$ is generated where i_x covers the branch b_x and $x = 1, 2, 3, ..., n$. This mapping can be represented as $P = I \longleftrightarrow B$. Since $I \longleftrightarrow B$ and $B \longleftrightarrow \mathcal{P}(S)$, it can be said that $I \longleftrightarrow \mathcal{P}(S)$ according to the law of transitivity. When m will be run for the input set I, $ran(I \longleftrightarrow \mathcal{P}(S))$ will be executed. Again, $ran(I \longleftrightarrow \mathcal{P}(S)) \cap S = \emptyset$

according to the conjunctive property of R and I. Since no statement is left after executing m against I, all possible behaviors are found using I. □

In order to determine two methods $m1, m2$ are feature wise similar, SBMF checks the signatures and generates the input set I using Algorithm 1. If the signatures matches, it executes $m1$ and $m2$ against I and obtains generated output sets $o1$ and $o2$ for $m1$ and $m2$, respectively. $m1$ and $m2$ are said to be feature-wise similar if $o1$ and $o2$ are the same set. It is possible to have two methods are functionally different but $o1$ and $o2$ are the same for I. Considering this issue Lemma 2 is derived.

```

int sum(int a, int b){
int result=a+b;
return result;
}

int multiply(int a, int b){
int result=a*b;
return result;
}
```

Listing 5.3: Input Data Set Selection

Lemma 2. *A single input value may cover all possible branches but may not separate functionally dissimilar methods.*

Proof. Assume that M is a set of methods and I is the input set. The negation of the lemma, $\forall m_1, m_2 \in M \bullet o_1 = o_2 \Rightarrow m_1$ and m_2 are functionally similar. Given that I covers all the branches of m_1 and m_2, and the respective output sets for I are o_1 and o_2. In order to proof the lemma, it is sufficient enough to show an example that disproves the negation following the existential quantification. Consider the code snippet shown in Listing 5.3 where two methods named *sum* and *multiply*

are declared. *sum* takes two integer numbers and return the summation of these numbers. *multiply* performs multiplication on two given integers. These methods are functionally different from each other. Assume that, an input set {(2,2)} is selected to check that *sum* and *multiply* are feature-wise dissimilar. The input set covers all the branches of the methods. If *sum* and *multiply* are executed for the input set, both method will produce the same output which is 2. Although these methods are functionally different, these are considered as similar methods due to producing the same output. Thus, it can be said that an input value may cover all branches but cannot differentiate functionally different at all. □

Considering the lemma, SBMF generates input dataset by applying boundary value analysis on each branch. For each branch, it generates three input values for a parameter which are the maximum, minimum and a randomly generated values based on the data type of the parameters. The reason behind using Boundary Value Analysis (BVA) is that BVA has been proposed to check the functionalities of a method.

5.4 Vocabulary Mismatch Problem

The effectiveness of a code search engine depends on the accuracy in representing user needs into query terms [56]. Proper transformation of information-needs into query terms helps to retrieve more relevant codes. Most of the code search engines provide a single text box to obtain a set of keywords that maps user query [26]. These keywords are matched against the collection index to retrieve relevant code snippets or software artifacts. However, if user provided keywords do not express the desired search topic or context properly, search engines fail to retrieve many relevant codes [96]. Sometimes, many irrelevant code fragments are shown as top ranked search results by search engines due to using irrelevant keywords as query terms. This problem is commonly known as vocabulary mismatch problem in

Information Retrieval (IR) [56], and it is also common in code search, because keywords are used to define user query [20].

5.5 Mathematical Model of Vocabulary Mismatch Problem

For a given query $q = \{t_1, t_2, t_3, ..., t_n\}$ and a set of methods $C = \{m_1, m_2, m_3, ..., m_i\}$, the set of all relevant methods for q is defined as

$$R := \{r \in C : Relevant(r, q)\}$$

$$\text{where, } R \subseteq C$$

For any term $t \in q$, *Vocabulary Mismatch Problem* is defined as the probability of t not appearing in a method m given that the method is relevant to q. The mismatch probability can be measured by taking the proportion of relevant methods in R that do not contain the term t. If the probability is denoted as $P(\bar{t}|R)$, then

$$P(\bar{t}|R) := \frac{|\{m \in R : t \notin m\}|}{|R|}$$

Here, $|R|$ is the cardinality of the set R, that is the number of relevant methods for the query q.

In order to solve the problem, SBMF uses Automatic Query Expansion technique. In this technique, each query term $t \in q$ is expanded with a set of extra terms, $T = \{t'_1, t'_2, ..., t'_n\}$ such that $P(\bar{t}|R)$ is reduced. More precisely, the set T should contains more relevant terms so that,

$$P(t|R) < P(t \vee t'_1 \vee t'_2 \vee t'_3 ... \vee t'_n|R)$$

$$= P(t|R) < 1 - P(\bar{t} \wedge \bar{t'_1} \wedge \bar{t'_2} \wedge \bar{t'_3} ... \wedge \bar{t'_n}|R)$$

$$= P(t|R) < \frac{|\{m \in M; t' \in T | t' \in m \bullet m\}|}{|R|}$$

$$= \frac{|\{m \in R : t \notin m\}|}{|R|} < \frac{|\{m \in M; t' \in T | t' \in m \bullet m\}|}{|R|}$$

SBMF constructs the set T by mining search log history as shown in Section 4.8. For each user query in the log history, SBMF checks the co-occurrences of the query terms in the query and search results. Later, score is assigned to each co-occurred term based on the number of queries that contain the term. For two terms - a, b and a collection of queries Q, the formula to calculate the score is defined as follows.

$$score(a, b) = |q \in Q : a \in q \wedge b \in q|$$

The set T is constructed by selecting top score terms for each query terms as shown in Algorithm 8. Later, the query is expanded with T to retrieve more relevant methods from the code base.

5.6 Hierarchical Clustering Problem

Usually, developers implement the same functionality in different ways. Implementations may differ from one technique to another in terms of different complexities such as time, memory, cyclomatic complexity, etc. A hierarchical relationship can be established among the terms found in the source codes of the implementations. To understand the relationship, an example code snippet is shown in Listing 5.4. The snippet contains four methods which are *mergeSortAscending*, *mergeSortDescending*, *bubbleSortAscending* and *bubbleSortDescending*. For a given array of integers, both *mergeSortAscending* and *bubbleSortAscending* sort the array in ascending order. Both methods perform the same task but the complexity of *mergeSortAscending*, $O(nlog(n))$ is less than the complexity of

bubbleSortAscending, $O(n^2)$. On the other hand, *mergeSortDescending* and *bubbleSortDescending* are functionally similar, because the methods sort an array of integers in descending order. However, the methods are different in terms of complexity as stated earlier. It can be seen that these four methods sort an array of integers where two of these sort in ascending order and other sort in descending order. If keywords are extracted from the names of the method, a hierarchical relationship can be found as shown in Figure 5.1.

```
1
2 void mergeSortAscending(int* A,int p,int r) {
3   if(r>p) {
4     int q;
5     q=(p+r)/2;
6     mergSortAscending(A,p,q);
7     mergSortAscending(A,q+1,r);
8     int n1=q-p+1;
9     int n2=r-q;
10    int L[n1+1];
11    int R[n2+1];
12    for(int i=0; i<n1; i++) L[i]=A[p+i];
13    for(int j=0; j<n2; j++) R[j]=A[q+1+j];
14    int i=0;
15    int j=0;
16    int n=0;
17    while(i!=n1 && j!=n2) {
18      if(L[i]>R[j]) {
19      A[p+n]=R[j];
20      j++;
21      } else {
22        A[p+n]=L[i];
23        i++;
24      }
25      n++;
```

```
26      }
27      while(j!=n2) {
28        A[p+n]=R[j];
29        j++;
30        n++;
31      }
32      while(i!=n1) {
33        A[p+n]=L[i];
34        i++;
35        n++;
36      }
37    }
38  }
39  void mergeSortDescending(int* A,int p,int r) {
40    if(r>p) {
41      int q;
42      q=(p+r)/2;
43      mergSortDescending(A,p,q);
44      mergSortDescending(A,q+1,r);
45      int n1=q-p+1;
46      int n2=r-q;
47      int L[n1+1];
48      int R[n2+1];
49      for(int i=0; i<n1; i++) L[i]=A[p+i];
50      for(int j=0; j<n2; j++) R[j]=A[q+1+j];
51      int i=0;
52      int j=0;
53      int n=0;
54      while(i!=n1 && j!=n2) {
55        if(L[i]<
56        R[j]) {
57          A[p+n]=R[j];
58          j++;
59        } else {
```

```
60            A[p+n]=L[i];
61            i++;
62          }
63          n++;
64        }
65        while(j!=n2) {
66          A[p+n]=R[j];
67          j++;
68          n++;
69        }
70        while(i!=n1) {
71          A[p+n]=L[i];
72          i++;
73          n++;
74        }
75      }
76  }
77
78  void bubbleSortAscending(int* A, int sz) {
79      for(int i=0;i<sz;i++){
80        for(int j=0;j<sz-1;j++){
81          if(A[j]>A[j+1]){
82            int temp=A[j];
83            A[j]=A[j+1];
84            A[j+1]=temp;
85          }
86        }
87      }
88  }
89
90
91  void bubbleSortDescending(int* A, int sz) {
92      for(int i=0;i<sz;i++){
93        for(int j=0;j<sz-1;j++){
```

```
94          if(A[j]<A[j+1]){
95            int temp=A[j];
96            A[j]=A[j+1];
97            A[j+1]=temp;
98          }
99        }
100     }
101 }
```

Listing 5.4: Sample Code to Demonstrate Hierarchical Clustering Problem

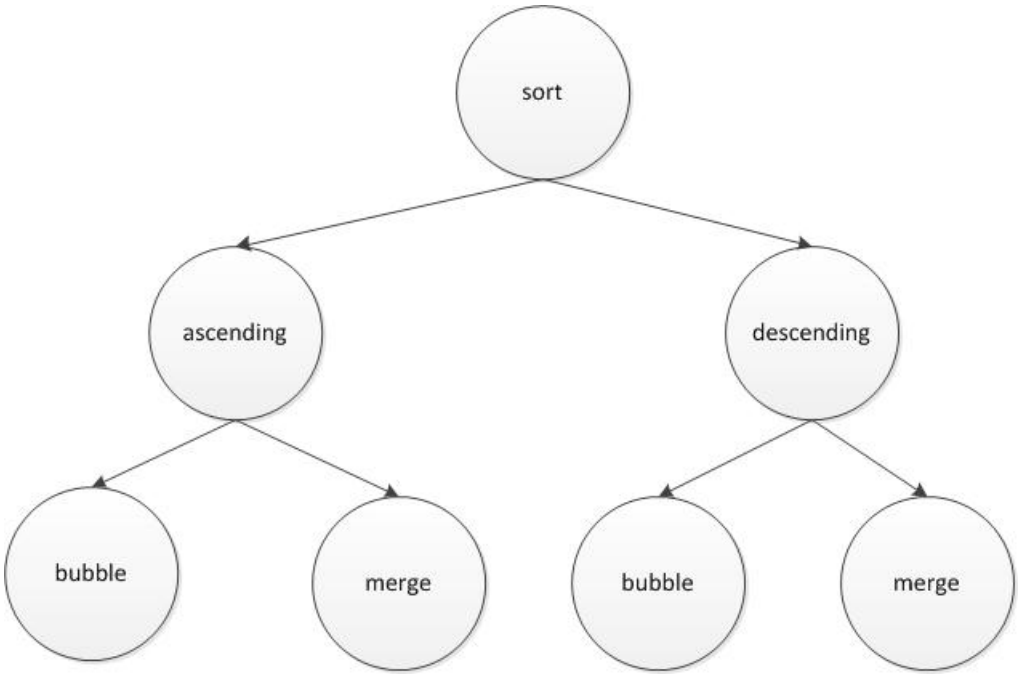

Figure 5.1: Hierarchical Relationship Among Terms for Listing 5.4

According to the Figure 5.1, the top term is sort because all the methods sort an array of integers. *ascending* and *descending* are the childs of *sort*, because ascending order and descending order are two types of sorting. Considering the relationship, the index terms for the methods are shown in Table 5.2. According to this hierarchical relationship, when a query contains the keyword *sort*, all the four methods should be retrieved. If the query contains *sort* and *ascending* keywords, $mergeSortAscending$ and $bubbleSortAscending$ should be retrieved. That means,

Table 5.2: Index Terms for the Methods in Listing 5.4

Method Name	Index Terms
mergeSortAscending	sort, ascending, merge
mergeSortDescending	sort, descending, merge
bubbleSortAscending	sort, ascending, bubble
bubbleSortDescending	sort, descending, bubble

if a query term matches any of the nodes in the hierarchical relationship, all the descendant nodes should be considered to retrieve functionally similar codes.

5.7 Mathematical Model of Hierarchical Clustering Problem

Assume that a finite set of methods is denoted by $M = \{m_1, m_2, m_3, ..., m_n\}$. In order to measure the similarity between a pair of methods, a function $sim(m, m')$ is defined, where $m, m' \in M$. m_1 and m_2 are said to be similar if the methods performs the same functionality. That is,

$$sim(m, m') = \begin{cases} 1 & \text{if } m \text{ and } m' \text{ are functionally similar} \\ 0 & \text{otherwise} \end{cases}$$

The similarity measurement function, $sim(m, m')$ is symmetric, that is

$$sim(m, m') = sim(m', m), \forall m, m' \in M$$

In order to perform cluster analysis, SBMF takes the pair (M, sim) as shown in Algorithm 4. The technique provides a partition $\Lambda = \{C_1, C_2, C_3, ..., C_N\}$ where $G_k, k = 1, 2, 3, ..., N$ is subset of M such that

$$C_1 \cup C_2 \cup C_3 \cup ... \cup C_N = M, C_i \cap C_j = \emptyset, i \neq j$$

Here, $C_1, C_2, C_3, ..., C_N$ of Λ are called clusters of functionally similar methods. A hierarchy of sub-clusters inside a cluster can be defined as a series of partitions, denoted by $\{\Lambda^i\}$ for $i = 1, 2, 3, ..., K$ if the following condition satisfies. For two partitions

$$\Lambda^i = \{C_1^i, C_2^i, ..., C_{N_i}^i\}, \Lambda^j = \{C_1^j, C_2^j, ..., C_{N_j}^j\}, i = 1, 2, 3, ..., K, i < j$$

Λ^i is a refinement of Λ^j. That is, for any member subset $C_k^i \in \Lambda^i$, there exists $C_k^j \in \Lambda^j$ such that C_k^i is the subset of C_k^j.

$$\forall C_k^i \in \Lambda^i, \exists C_k^j \in \Lambda^j \bullet C_k^i \subseteq C_k^j$$

Such a series of partitions is called a hierarchy of clusters. In order to construct the hierarchy of clusters, the step *Clustering Similar Methods* of SBMF is presented in the previous chapter (Section 4.5). In the step, a given set of functionally similar methods C is decomposed into a set of clusters R based on time and memory space complexities. That is, $R \subseteq C$ and $\forall r : R \cdot$ ($\forall p, q : r \cdot O(p) = O(q)$ and p, q are functionally similar). Later, terms are extracted from R to identify the hierarchical relationship. Common terms among the clusters are moved to upper level and rare terms are placed at the bottom of the hierarchy.

5.8 Summary

In this chapter, three problems - *Method Name Inconsistency*, *Vocabulary Mismatch Problem* and *Hierarchical Clustering Problem* are identified that reduce the effectiveness in code search. $MethodNameInconsistency$ is responsible to index a method against irrelevant terms and it is modeled mathematically in Section 5.3. SBMF solves it by indexing functionally similar methods against common and relevant terms. A theoretical analysis is given to justify the correctness of

the solution. *Vocabulary Mismatch Problem* decreases the matching probability between query terms and index terms as shown mathematically in Section 5.5. SBMF expands the query with relevant terms to solve the problem. The procedure of selecting relevant terms, is also mathematically explained. SBMF checks hierarchical relationship among a set of index terms. The relationship is theoretically discussed in Section 5.7.

In essence, all the problems and the respective solutions are mathematically modeled in this chapter. A theoretical analysis is also provided to justify how the proposed technique, SBMF resolves the problems. The next chapter discusses a comparative result analysis of SBMF with the experimental setup.

Chapter 6

Implementation and Result Analysis

This chapter presents the experimental analysis of the proposed technique, Similarity Based Method Finder (SBMF). For the experimental evaluation, a prototype of the technique was implemented in Java programming language. As discussed in the previous chapter, *vocabulary mismatch problem* in query level decreases the effectiveness of a code search engine. To solve that problem, SBMF mines user query logs and selects expansion terms for a given user query. In order to evaluate the expansion process, 22 real life user queries and one year search logs of Koders were used in the experiment. SBMF was compared to an existing technique named Thesaurus-based Automatic Query Expansion [20] in the experiment. *Method name inconsistency* also reduces the effectiveness in code search as mathematically shown in the previous chapter. To evaluate the effectiveness of SBMF in retrieving codes through solving the problem, an experiment was conducted. 50 open source projects were selected for the experimentation. 195 queries representing 9 different features were executed by the prototype of SBMF. For the comparative analysis, same queries were also run on Sourcerer [11] that supports Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS).

In this chapter, a brief explanation regarding the implementation environment and dataset information for the experiment are provided and a comparative analysis is also explained in details.

6.1 Implementation Details

As said earlier, a prototype of SBMF was implemented for the experimental analysis. The tools and frameworks used for the implementation, are given as follows.

- **Juno[1]:** An open source Integrated Development Environment (IDE) that facilitates developing software in Java programming language. The technique SBMF was implemented using this IDE. Writing, building, managing, and running source code of SBMF are done with the assistance of it.

- **Java Parser[2]:** It is an open source Java library for parsing Java source code. It takes the source code of a java project including all dependencies as input and parses the code by constructing Abstract Syntax Tree. Next, it traverses the tree to provide information about the entities in the source code. For example, class, method body, method signature, invoked method statements, field declaration, library declaration, etc. The library was used in the prototype to parse source code of the experimental datasets.

- **Apache Lucene[3]:** It is a popular open source search engine infrastructure. Apache software development community published the jar file of the library. In order to construct the index of methods, Lucene was integrated to the prototype. The latest stable version of Lucene, that is lucene-core-3.6.2.jar was used in the prototype.

[1]https://eclipse.org/juno/
[2]https://github.com/javaparser
[3]https://lucene.apache.org/

- **Luke**[4]**:** An open source Lucene client named Luke was used in the prototype. Luke supports boolean query from the user and it executes the query on the Lucene index. It also assists to display search results. To get user query and retrieve relevant methods from the index, Luke was plugged in the prototype of SBMF

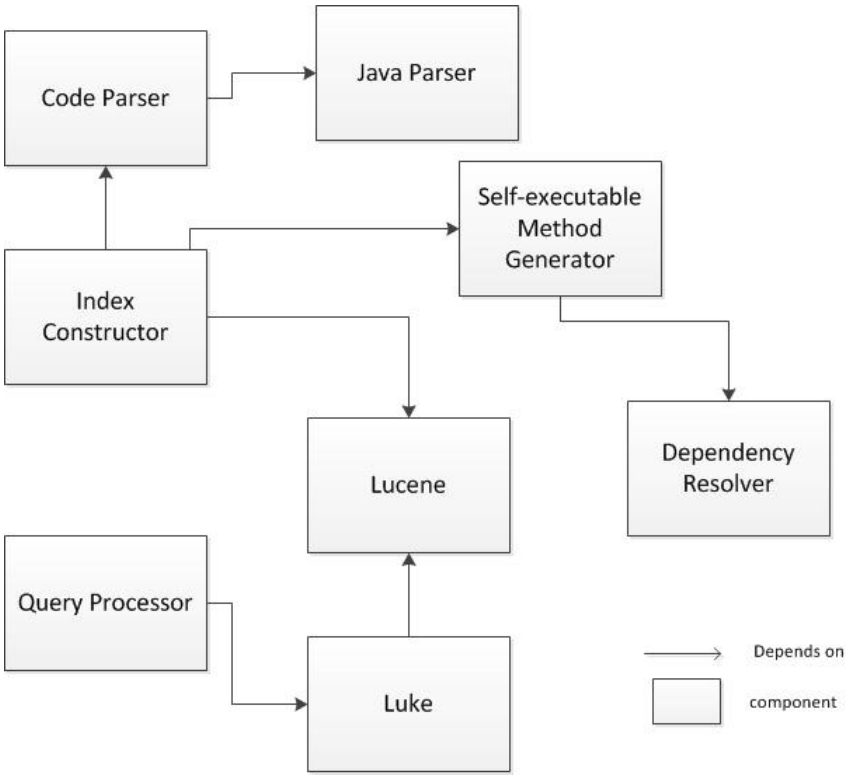

Figure 6.1: Conceptual Architecture of the Prototype

Conceptual Architecture of the Prototype

The conceptual architecture of the developed prototype is shown in Figure 6.1. There are eight components found in the prototype of SBMF and these are *Code Parser, Java Parser, Index Constructor, Self-executable Method Generator, Query Processor, Lucene, Dependency Resolver,* and *Luke.* For a given code base, *Code Parser* parses source codes and generates Abstract Syntax Tree. It depends on

[4]https://github.com/DmitryKey

Java Parser for the tree construction. The tree is then traversed to find declared methods in the source code. *Index Constructor* delivers the methods to *Self-executable Method Generator* that converts the methods into self-executable code fragments. The component depends on *Dependency Resolver* to resolve function call, data and library dependencies. All the self-executable methods are then indexed by *Lucene*. *Query Processor* takes query from the user and searches relevant self-executable methods in the index through *Luke*. Relevant self-executable methods are retrieved and displayed by *Luke* as the search results against the query. The source codes of the prototype is available in Github [5].

6.2 Experimental Analysis on Query Expansion

Vocabulary mismatch problem causes the search engines to retrieve irrelevant codes from a code base as stated earlier. In order to solve the problem, SBMF mines search engine usage logs. It selects relevant terms for query expansion. An experimental analysis is provided in this section to evaluate the query expansion process of SBMF. The section discusses the experimental setup, dataset, and metric that are used in the experiment. Besides, a comparative result analysis is also provided in the section.

6.2.1 Experimental Setup

This section outlines the software and frameworks required for the experimental analysis. In order to evaluate SBMF, the developed prototype was used in the experiment. For the comparative result analysis, an existing technique named Thesaurus-base Automatic Query Expansion (TBAQE) [20] was implemented using C# programming language. Some other tools were also used in the experiments. These are addressed as follows.

[5]https://github.com/rifatbit0401/sbmf

Table 6.1: Sample Queries with Frequency

#	Query	Number of Occurrences
1	apache	411
2	audio	360
3	awt	189
4	dao	390
5	data source	5
6	data structure	42
7	date	1840
8	files	2164
9	ftp	1865
10	hibernate	640
11	huffman	967
12	image	1693
13	jsp	423
14	list	1241
15	listener	208
16	log	879
17	lucene	383
18	parser	1449
19	sort	2402
20	spring	381
21	test	1537
22	webservice	297

- **MSSQL[6]:** A database engine used to store query logs.
- **Visual Studio 2012**: An Integrated Development Environment for C# used to implement TBAQE.
- **WordNet[7]:** A natural language thesaurus adopted in this experiment for query expansion.

6.2.2 Dataset Selection

One year long usage log of a commercial code search engine named Koders was collected from [19]. These dataset was fed into SBMF to identify topic specific

[6]https://www.microsoft.com/en-us/cloud-platform/sql-server
[7]https://www.wordnet.princeton.edu/wordnet/download/

query terms for query expansion. 22 real user queries were obtained from Koders. A summary of these queries are presented in Table 6.1 where the second column (Query) contains all the original query keywords, and third column (Number of Occurrences) depicts the frequency of the queries within a year.

15 subjects were selected to evaluate the relevance of the search results. Five of them were senior software engineers and rest ten were masters students. The reason behind choosing students in this study is that they can play important role in software engineering experiments [112]. All the experimental datasets and source code are available in *http://tinyurl.com/zlfwbjn*.

6.2.3 Experimental Metric

To compare the effectiveness between SBMF and TBAQE, a metric named Precision at 10 (P@10) was used. The reason behind choosing this metric is that the amount of open source projects are increasing rapidly and many relevant code fragments or software artifacts are retrieved by the search engines. Currently, developers are more interested to find the desired results within top 10 retrieved code snippets instead of going through all the fetched results. The metric P@10 is defined as the number of relevant results from the first 10 retrieved code snippets or software artifacts. If x be the set of relevant code snippets from the first 10 retrieved results, P@10 can be defined as follows.

$$P@10 = \frac{|x|}{10} \tag{6.1}$$

The nominator of the equation, $|x|$ represents the cardinality of the set x that is the number of relevant results from the top 10 retrieved code snippets.

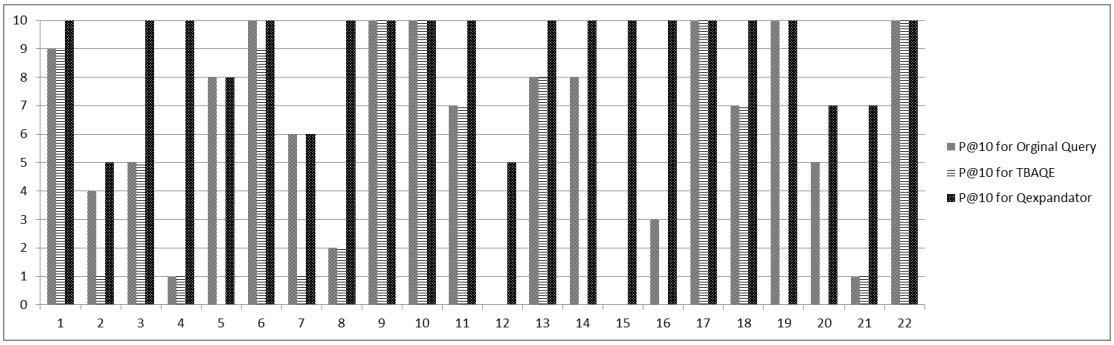

Figure 6.2: Precision@10 Analysis

6.2.4 Comparative Result Analysis

For the comparative result analysis, queries in the experimental dataset were expanded by QExpandator and modified queries were run in Google. Relevance of the retrieved results was judged by the subjects. Same procedure was followed for the existing technique TBAQE where queries were expanded using WordNet. All the queries with the respective expanded forms are shown in Table 6.1. The second column of the table contain all the sample queries. The third and fourth column hold the queries expanded by TBAQE and SBMF respectively.

A comparative result analysis is depicted in Table 6.2. For each query, corresponding values of P@10 with respect to original query, TBAQE, and SBMF are shown in third, fourth, and fifth columns respectively. While analyzing the results it is seen that 11 queries cannot be expanded by the existing technique TBAQE. The reason is that these queries contain technical terms and names of different frameworks which have no synonyms in the WordNet. On the other hand, QExpandator expands these queries by adding conceptually related terms through analyzing search log history. For example, query#3 contains "awt" which is basically a Graphical User Interface library provided in Java language. So this term is expanded with the most relevant expanded term named "GUI".

Figure 6.2 depicts the value of P@10 for each query with respect to no expansion (original query), TBAQE, and QExpandator. Here X-axis denotes the query# as shown in Table 6.2 and Y-axis denotes the value of P@10. According

to the figure, TBAQE decreases P@10 for 31.82% of total user queries whereas SBMF increases P@10 for almost 63.64% user queries and it does not reduce P@10 for the other queries. The reason for such behavior of TBAQE is that it adds synonyms as additional terms to the original query and these terms express different semantic meanings from technical point of view. The expanded query then represents information needs that are different from the original query. For example, in query#19 "sort" keyword is used to get code snippets that order items in a list in specific order. One of the synonyms of sort is "sieve" which is used in the expanded query by TBAQE. "sieve" represents a prime number generator algorithm in programming context and it has no similarity with the keyword "sort". As a result, the expanded query suffers from reduced P@10. Conversely, "bubble" keyword is appended to the query sort by QExpandator and it is consistent with the search topic because bubble sort is a special type of sorting algorithm. Due to adding context and topic specific terms, QExpandator increases the value of P@10 or keeps the same if improvement is not possible.

Ambiguity in query terms are found in several queries such as query#2, query#14, query#15, query#16, and query#21 as shown in Table 6.2. Usually ambiguous terms express different semantic meanings in different contexts. For example, in query#16 "log" keyword has three semantic meanings which are logarithmic function, an element of a tree, and observing execution traces of a program. TBAQE uses "logarithm" as synonym of "log" keyword and forces the search engine to retrieve logarithmic functions which reduces P@10. This is because the search intent is to find example code of tracking execution behavior. However, SBMF adds "log4j" as expanded term which is relevant to capturing and working with a program execution sequence. The technique handles ambiguous terms by adding search intent specific keywords and increases the effectiveness of search engine.

Although SBMF improves P@10 for 63.64% of total user queries by adding search topic specific keywords, and for the rest 36.36% queries, it cannot increase

Table 6.2: Comparative Result Analysis on Query Expansion

#	Query	P@10 for Orginal Query	P@10 for TBAQE	P@10 for Qexpandator
1	apache	9	9	10
2	audio	4	1	5
3	awt	5	5	10
4	dao	1	1	10
5	data source	8	0	8
6	data structure	10	9	10
7	date	6	1	6
8	files	2	2	10
9	ftp	10	10	10
10	hibernate	10	10	10
11	huffman	7	7	10
12	image	0	0	5
13	jsp	8	8	10
14	list	8	0	10
15	listener	0	0	10
16	log	3	0	10
17	lucene	10	10	10
18	parser	7	7	10
19	sort	10	0	10
20	spring	5	0	7
21	test	1	1	7
22	webservice	10	10	10

the value of P@10. The reason is that most of these queries contain keywords that represent the information needs properly. Conversely, TBAQE reduces P@10 for 31.82% user queries due to adding only synonyms that do not consider search context properly.

In essence, SBMF improves P@10 by 29.1% on an average that leads to the ultimate P@10 value 90% for each user query. Moreover, the technique does not hurt or reduce the value of P@10 for any query. Conversely, TBAQE reduces the P@10 for 31.82% user queries and no improvement is found in any user query for this technique. QExpandator performs better than TBAQE because it adopts search topic and context specific keywords for query expansion instead of synonyms that have different meanings in various contexts.

Table 6.3: Expanded Queries

#	Query	Query Expanded by Theasaurus	Query Expanded by SBMF (the most relevant term is added)
1	apache	apache	(apache OR tomcat)
2	audio	(audio OR sound)	(audio OR encoder)
3	awt	awt	(awt OR GUI)
4	dao	dao	(dao OR jdbc)
5	data source	(data OR information) AND (source OR seed OR germ OR reference OR beginning OR origin OR root OR rootage OR reservoir OR generator OR author OR informant)	(datasource OR connection)
6	data structure	(data OR information) (structure OR construction)	(data structure OR algorithm)
7	date	(date OR appointment OR engagement OR escort OR see)	(date OR format)
8	files	files	(files OR class)
9	ftp	ftp	(ftp OR server)
10	hibernate	hibernate	(hibernate OR jpa)
11	huffman	huffman	(huffman OR coding)
12	image	(image OR effigy OR simulacrum OR picture OR icon OR ikon OR persona OR prototype OR paradigm OR epitome OR trope OR figure OR double OR visualize OR visualiseOR envision OR project OR fancy OR see)	(image OR attribute)
13	jsp	jsp	(jsp or java)
14	list	(list OR tilt OR inclination OR lean OR leaning OR listing OR name OR number OR heel)	(list or util)
15	listener	(listener OR hearer OR auditor OR attender)	(listener or event)
16	log	(log OR logarithm OR backlog OR lumber)	(log OR log4j)
17	lucene	lucene	(lucene OR solr)
18	parser	parser	(parser OR dom)
19	sort	(sort OR kind OR form OR variety OR sorting OR classify OR class OR assort OR separate OR screen OR sieve)	(sort OR bubble)
20	spring	(spring OR leap OR leaping OR saltation OR bound OR bounce OR give OR springiness OR fountain OR outflow OR outpouring OR springtime OR resile OR rebound OR recoil OR reverberate OR ricochet OR jump OR form)	(spring OR framework)
21	test	(test OR trial OR run OR tryout OR examination OR exam OR quiz OR prove OR try OR examine OR essay OR screen)	(test OR Junit)
22	webservice	webservice	(webservice OR SOAP)

6.3 Experimental Analysis on Retrieval Effectiveness

In order to evaluate the retrieval effectiveness of SBMF, an experiment was conducted on 50 open source software projects. 195 real life user queries representing 9 different features were also used in the experiment. All the queries were collected from [20]. For the comparative result analysis on retrieval effectiveness, an existing code search engine named Sourcerer [11] was used as the implementation of KBCS and IDCS. The environmental setup, experimental metrics and comparative result analysis of the experiment are discussed in the section.

6.3.1 Environmental Setup

For the experimental analysis, the prototype of SBMF was used in the experiment. Sourcerer implementation was also taken from [11]. The experiment was conducted in a single machine. The configuration of the machine is outlined as follows.

- **Processor:** Intel(R) Core(TM) i5 -2430M CPU @2.40GHz
- **RAM:** 4GB
- **Operating System:** Windows 7 Ultimate
- **System Type:** 32-bit Operating System

6.3.2 Dataset Selection

In order to perform experimental analysis, 50 open source projects from SourceForge[8] were selected. Fraser and Arcuri showed that these projects are statistically sound and representatives of open source projects [113].

[8]https://sourceforge.net/

Table 6.4: Selected Functionalities with Frequency

#	Functionality	# methods	# queries
1	decoding String	13	20
2	encrypting password	3	27
3	decoding a URL	3	21
4	generating MD5 hash	3	16
5	rotating array	2	25
6	resizing image	3	25
7	scaling Image	3	19
8	encoding string to html	2	6
9	joining string	47	36

A set of features were selected from the existing works in code search [7, 17, 114, 20] as shown in Table 6.4. On the other hand, to evaluate the proposed technique, a set of queries was selected from [7]. Here, each query is related to a particular functionality shown in Table 6.4. The table also presents the frequency of methods that implement a feature of experimental dataset. For example, 13 methods were found in the repository that implement feature#1 *decoding string*, and 20 queries were submitted for the feature as shown in the table. 15 subjects were employed to identify relevant methods for the functionalities. Among the subjects, five of them were senior Java developers and other ten were masters students. The reason of choosing students in this study is that they can play important role in the software engineering experiments as suggested in [112]. All the experimental datasets are available in *http://tinyurl.com/zdqmoqz*.

6.3.3 Experimental Metrics

Usually, recall and number of retrieved methods are used to measure the effectiveness of a code search engine [14]. These metrics were used to measure the retrieval effectiveness of SBMF. Other than that, two metrics - *feature successfulness* and *self-executability* were used to measure the quality of the retrieved code snippets. All the experimental metrics are explained as follows.

- **Recall:** This metric is commonly used to evaluate the effectiveness of a code search engine. So, the metric was used for the experimental analysis. In the experiment, recall for a query q is defined in the following.

$$recall_q = \frac{|retrieved\ methods \cap relevant\ methods|}{|relevant\ methods|} \tag{6.2}$$

That means, recall is the proportion of the number of relevant methods retrieved and total number of relevant methods for a given query. In the experiment, a set of queries was collected for each feature. Average recall was used to perform the feature-wise recall analysis. That is, the recall of a feature is the average recall of all the queries of the feature. Let $Q = \{q_1, q_2, q_3, ..., q_n\}$ be the set of all queries for a feature f. The recall of the feature is as follows.

$$recall_f = \frac{\sum_{i=1}^{n} recall_{q_i}}{n} \tag{6.3}$$

- **Number of Retrieved Methods (NRM):** In order to know the number of methods retrieved for a given query, this metric was used in the experiment. Recall and NRM jointly provide a notion on the number of relevant and irrelevant methods retrieved. For a query q, NRM can be defined as follows.

$$NRM_q = |retrived\ relevant\ methods \cup retrieved\ irrelevant\ methods \tag{6.4}$$

In the experiment, a set of queries for each of the experimental features was evaluated. In order to analyze the number of methods retrieved for each feature, average number of retrieved methods was used in the experiment.

If $Q = \{q_1, q_2, q_3, ..., q_n\}$ is the set of queries for a feature f, the number of retrieved methods NRM_f for f can be calculated using following equation.

$$NRM_f = \frac{\sum_{i=1}^{n} NRM_{q_i}}{n} \tag{6.5}$$

- **Feature Successfulness:** Developers look for code snippets to implement a particular feature. Although the index has relevant code fragments for a given feature, any of these codes may not be retrieved. It happens due to irrelevant index terms, or ambiguous query terms. A search engine will be effective if it can retrieve these codes by solving the problems. To know the number of features for which the code search engine can retrieve relevant codes, *Feature Successfulness* is introduced. If Q be the set of queries for a feature f and R be the set of relevant code fragments in the index, *feature successfulness* denoted by FS for f can be defined as follows.

$$FS(f) = \begin{cases} 1 & \text{if } \exists q \in Q : retrievedMethods(q) \cap R \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

  That is, a feature is said to be successful if at least one relevant method is retrieved that implements the feature f.

- **Self-executability:** Developers use a code search engine to retrieve relevant code snippets and reuse these in the development context. It takes additional time and effort to plug the code snippets and make executable. The reason is that most of the cases, code search engines do not provide code snippets with all dependencies and developers need to resolve those manually. Considering the matter, a metric named *self-executability* is introduced. The metric checks whether the retrieved code snippet is executable or it require manual intervention to make it workable. A code fragment is said to

be self-executable if developers can copy-paste it in the development context and the code fragment is executable. In the experiment, the percentage of retrieved self-executable methods for each feature was used for the comparative result analysis. Let $Q = \{q_1, q_2, q_3, ..., q_n\}$ is the set of queries for a feature f and X is the set of retrieved methods for all the queries in Q. The formula to calculate the percentage of retrieved self-executable methods $PRSEM_f$ for f is as follows.

$$PRSEM_f = \frac{|\{x \in X : x \textit{ is self executable}\}|}{|X|} \times 100 \tag{6.7}$$

Here, nominator is the number of self executable methods retrieved for all queries representing f. The denominator is the number of method retrieved for the queries that are used for f.

6.3.4 Comparative Result Analysis

For comparative result analysis, SBMF was run on the experimental datasets and the relevance of retrieved methods were checked for each user query. Moreover, Sourcerer which supports KBCS and IDCS, was also run on the same datasets. In addition, search results obtained from the Sourcerer are compared to SBMF. Detailed result analysis with respect to each of the experimental metrics is discussed as follows.

Recall Analysis

Figure 6.3 depicts a comparative recall analysis among SBMF, KBCS, and IDCS where X-axis denotes the feature no. as shown in Table 6.4 and Y-axis represents the measured recall. Here recall for each of the features was calculated using Equation 6.3. For feature #1 (Decoding String), approximately 15% recall is shown in Figure 6.3 for both KBCS and IDCS whereas 100% recall is found for

SBMF. There are 13 methods in the repository that implement the feature. Among these, two methods are found which contain keywords *decode* and *string* in method name and parameter respectively. As a result, these methods are retrieved by both KBCS and IDCS. However, these techniques cannot retrieve other 11 methods because signatures of these methods do not contain any term related to *decode*. While analyzing the source code of these methods, it is seen that the bodies of these methods use third party APIs such as *URLDecoder.decode(String, String), Hex.decode(String), Base64.decode(base64), etc.* to implement the feature. SBMF takes terms from API call statements and indexes against the terms to provide example codes regarding API usage. So, it retrieves all the 13 methods.

For feature #2 (Encrypting Password), IDCS cannot find any methods but 66.67% and 33.3% relevant methods are retrieved by SBMF and KBCS respectively as shown in Figure 6.3. To get the methods that implement this feature, the following query is provided to IDCS.

name:(encrypt) AND return:(String) AND parameter:(String)

Although there is a single method found in the code base that has *encrypt* keyword in its name but does not have *String* in its parameter. So, IDCS cannot obtain this method. However, KBCS retrieves the method because query keyword matches with the method name. On the other hand, SBMF retrieves one more method having signature *crypt(String strpw,String strsalt)*. The reason is that *encrypt* and *crypt* both express the same intent as detected by the query expansion part of SBMF.

There are 3 relevant methods in the experimental projects that implement feature #3 (Decoding a URL). According to Figure 6.3, only a single method is retrieved by SBMF that produces recall 33.33%. On the contrary, KBCS and IDCS cannot retrieve any method related to the feature. This is because no method

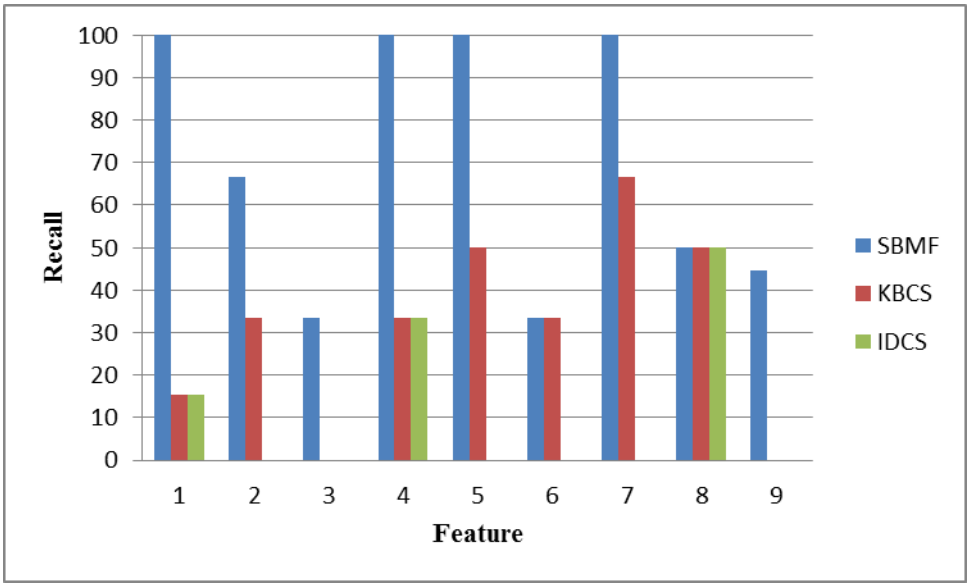

Figure 6.3: Recall Analysis

contains *decode* and *URL* simultaneously in the signature. Although one of these methods named *getPath* does not provide any semantic information representing the feature, it invokes a library method - *URLDecoder.decode(path, "UTF-8")* which implements the feature. SBMF considers the invocation statement for getting more relevant terms and thus, retrieves this method. Two other methods cannot be retrieved by SBMF due to finding no structural similarity among these and no keywords representing the feature.

According to Figure 6.3, 100% recall is obtained for SBMF, and 33.33% for KBCS and IDCS individually with respect to feature #4 (Generating MD5 hash). It is clear that SBMF has higher recall than other two approaches. The reason is that most of the methods implementing this feature do not have proper names to represent the intents. There are five methods relevant to this feature and only one method is found having name consistent with the feature. KBCS and IDCS fail to retrieve all these methods because both techniques extract terms from individual method and do not consider appropriateness of the terms. However, SBMF finds that these methods are semantically similar. These methods are indexed under common terms. As a result, when user query matches with one of these methods,

other three methods are also retrieved with this.

For features #5, #6 and #7, IDCS cannot retrieve any method from the code base used in this experiment. The reason is that appropriate parameter type is not determined in the user queries used for these features. However, KBCS shows 50%, 33.33%, and 66.67% recall for features #5, #6 and #7 respectively. On the other hand, SBMF shows 100% for features #5 and #7, and 33.33% for feature #6 as illustrated in Figure 6.3. For feature #5, two relevant methods are found which named as *transpose* and *rotate* correspondingly. These two methods are feature-wise similar. SBMF detects the similarities and indexes the methods under common terms - *rotate* and *transpose*. On the other hand, KBCS does not check similarity, and analyzes each method individually during indexing. So, only *rotate* method is retrieved by KBCS. For feature #7, SBMF retrieves one more method than KBCS because this method does not contain any term related to *image* but it uses a field of type *Image*. SBMF considers this usage since scaling operation is performed on this field by the method, and adds additional term *Image* against the method.

SBMF, KBCS, and IDCS show equal performance for feature #8 (Encoding String to HTML) in terms of recall. However, 50% relevant methods cannot be retrieved because no HTML keyword is found in these method.

Only SBMF is able to retrieve twenty one relevant methods whereas other techniques cannot fetch a single method for feature #9 (Joining String). Here, SBMF outperforms than KBCS and IDCS because it identifies many structurally similar methods which have different names but all these methods perform string concatenation. Among these, several methods are found which have proper keywords in their body. These keywords are attached to the term list of each similar method by SBMF. As a result, these are indexed under common appropriate terms and all the methods are retrieved simultaneously. However, other 26 relevant methods cannot be retrieved since no signature matching is found among these.

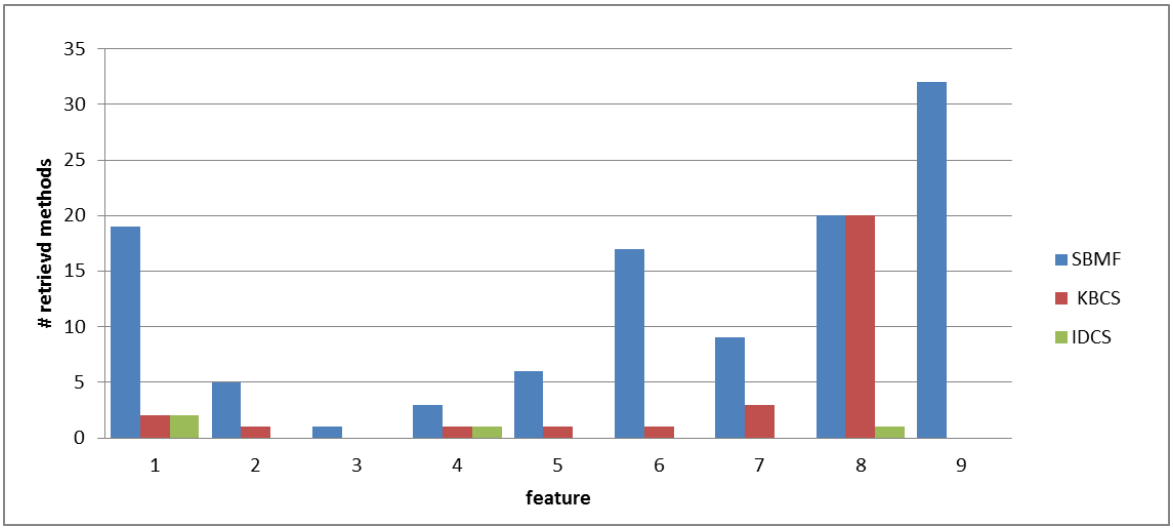

Figure 6.4: Number of Retrieved Methods

Number of Retrieved Methods (NRM) and Feature Successfulness Analysis

As NRM is an important measure to perceive recall of a search engine, a comparative result analysis with respect to NRM is shown here. A bar diagram is shown in Figure 6.4 depicting feature-wise NRM by SBMF, KBCS, and IDCS. According to the diagram, SBMF retrieves more methods than KBCS and IDCS because of adding common terms to each method.

Although IDCS produces better precision than KBCS and SBMF, it cannot retrieve a single method for some features (such as features #2, #3, #5, #6, #7, #9). The reason is that user queries do not have proper parameter type or return type. This scenario is common when developers have little or no knowledge about the implementation of a feature. KBCS and SBMF mitigate the problem by retrieving more relevant methods and adopting free text search. In order to determine whether a feature is successful or not, a metric named *feature successfulness* has been introduced. A feature is said to be successful if at least one relevant method is retrieved that implements the feature as defined in Equation 6.6. Figure 6.5 presents the number of successful features among SBMF, KBCS, and IDCS. According to this figure, SBMF is successful for all the 9 features whereas 7 and 3

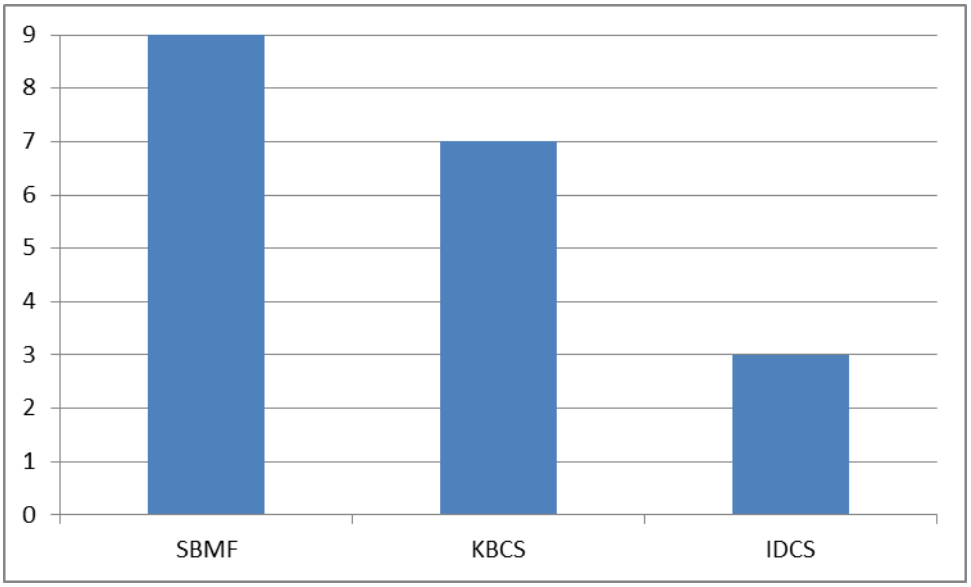

Figure 6.5: Feature Successfulness Analysis

successful features are found for KBCS and IDCS respectively. This measure provides a notion that having higher precision is not effective if number of successful feature is low. In addition, improving recall increases the chances of having higher number of successful feature. For this reason, SBMF performs better than KBCS and IDCS.

Self-executability Analysis

Figure 6.6 depicts the comparative result analysis among SBMF, KBCS, and IDCS in terms of self-executability. The Y-axis denotes the percentage of retrieved methods that are self-executable. The X-axis denotes the experimental features. For each feature, the percentage of self-executable methods is calculated using Equation 6.7. According to the figure, it can be seen that all the methods retrieved by SBMF are self-executable. This is because SBMF resolves method call, data, library dependencies to make a method self-executable. On the other hand, KBCS and IDCS considers a method as a plain text document and do not check any dependency while constructing the index. So, both methods cannot provide any self-executable method for features #2, #3, and #6. For feature #4, all the

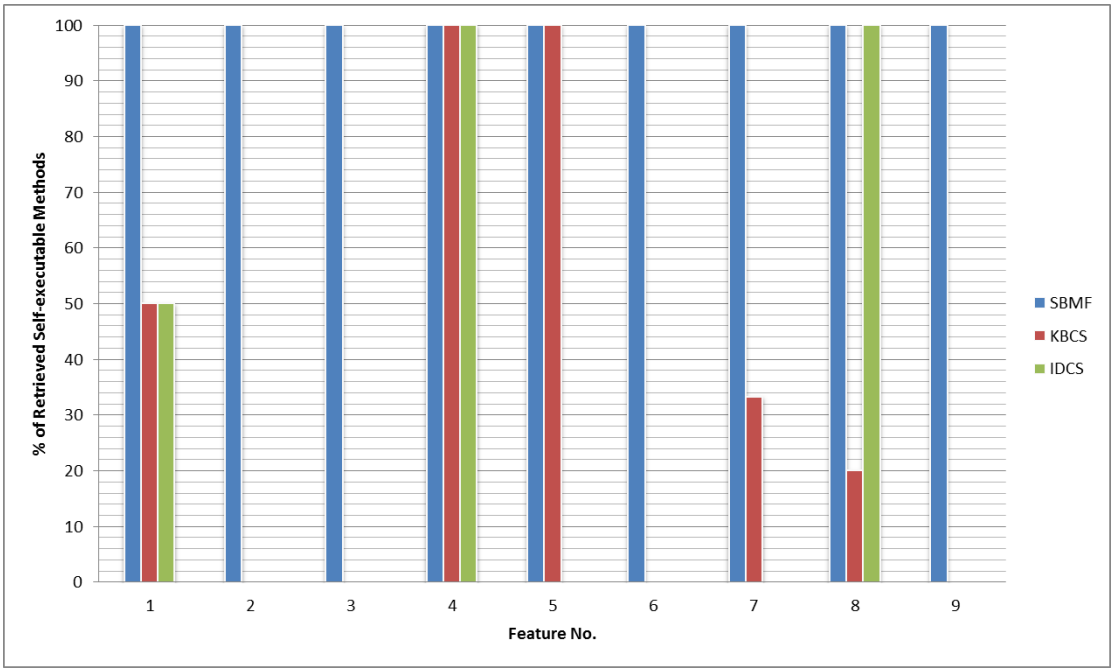

Figure 6.6: Self-executability Analysis

techniques show 100% in retrieving self-executable methods. Methods for this feature, do not have any dependency on other library, field, or method. So, these methods are directly pluggable to the desired development context. For other features, few methods are found as self-executable code fragments. These methods do not have any external dependency. Since KBCS and IDCS do not resolve any external dependency, manual effort is required to make methods executable in the development context. On the other hand, SBMF indexes methods with the dependencies required for the execution. So, developers can directly copy-paste the codes retrieved by SBMF.

6.4 Summary

This chapter presents the experimental analysis of SBMF to understand its effectiveness in code search. The first experiment was conducted to evaluate the query expansion process of SBMF. The experimental setup and test beds are discussed in the chapter. The result analysis of the experiment shows that SBMF improves

precision at 10 by 48.6% more than the existing technique TBAQE. The reason is that SBMF selects topic specific terms for query expansion instead of synonyms that TBAQE employs. The second experiment was performed to evaluate the effectiveness in retrieving relevant code. Section 6.3 describes the steps of the experiment and comparative result analysis. It is seen from the analysis that on an average, SBMF produces 38% and 58% more recall than KBCS and SBCS respectively due to indexing feature-wise similar methods against relevant terms. On an average, SBMF retrieves 66.29% and 72.22% more self-executable methods to satisfy all the experimental features in comparison with KBCS and IDCS, respectively. These results support that SBMF retrieves more relevant methods than the existing techniques. Thus, the technique improves the effectiveness in code search. The next chapter concludes the thesis with some future directions.

Chapter 7

Discussion and Conclusion

The effectiveness of a code search engine is reduced if functionally similar code fragments are not indexed under common proper terms. Developers are deprived of many relevant codes that are reusable in their development contexts. In this research, a technique named Similarity Based Method Finder (SBMF) has been proposed to improve the effectiveness in code search. The technique executes a given collection of methods to check the feature-wise similarity. It constructs clusters of methods based on the functional similarities. To index a cluster, the most frequently occurred terms are selected as index terms. The user query is expanded by adding topic and context specific terms. The comparative result analysis shows that SBMF produces 48.6% more precision at 10 than an existing technique named Thesaurus-Based Automatic Query Expansion (TBAQE) in expanding user queries. In addition, on an average, SBMF retrieves 38% and 58% more relevant methods than two existing techniques - Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS). This chapter describes SBMF in essence with its achievement in retrieving more relevant methods. Moreover, several threats to validity and future direction of this work are also discussed in this chapter.

7.1 SBMF: The Proposed Technique to Improve Effectiveness in Code Search

SBMF comprises six steps which are *Self Executable Method Generation, Method Signature Regeneration, Input Dataset Generation, Clustering Similar Methods, Proper Term Selection, Index Construction* and *Query Formulation*. The step *Self Executable Method Generation* parses all the methods from a given code base. It converts the methods into self-executable code fragments by resolving method call, data and library dependencies. *Method Signature Regeneration* redefines the signature of each self-executable method to explicitly define the parameters and return type. To determine all possible behaviors of a method, an input dataset is generated in the step *Input Dataset Generation*. After that, all the self-executable methods are clustered based on the feature-wise similarities. To detect the similarities, signature of the methods are checked. Methods having the same signature, are executed against an input dataset which is generated by *Input Dataset Generation*. The generated output sets are checked to cluster the methods. Methods that produce the same output set for the same input set, are added to the same cluster. To index the clusters, the step *Proper Term Selection* extracts terms by tokenizing and stemming textual words found in the methods. Most frequently appeared terms in a cluster are used to index the cluster. *Index Construction* is responsible to index the methods against the proper terms. In order to expand a user query, the step *Query Formulation* uses search log history to identify conceptually similar words. It converts each query into document vector where each term in the vector corresponds to the query term. Conceptually similar terms are identified by employing Jaccard similarity on the co-occurrence of document terms. A term-term matrix is constructed to store all the similar terms in the same row in descending order of the similarity score. Finally, for a given user query, each query term is expanded by adding top scored terms from the matrix.

7.2 Discussion of the Results

In order to assess the query expansion process of SBMF, an experiment was conducted on 22 real life user queries collected from Koders [9]. An existing technique named TBAQE [20] was used for the comparative result analysis. According to the result analysis, SBMF has produced 48.6% more improvement in precision at 10 (P@10) than TBAQE. Moreover, on an average, it has increased the value of P@10 from 60.9% to 90% for each query. The reason is that, instead of using synonyms, it has added context specific keywords as expanded terms.

Another experiment was performed to evaluate the retrieval effectiveness of SBMF. For the experimental analysis of the technique, fifty open source projects were selected to build the code base and nine features were chosen to generate queries. To compare the results with two types of existing techniques named KBCS and IDCS, Sourcerer [11] was used. The reason is that Sourcerer supports both KBCS and IDCS. While analyzing the results, it has been seen that SBMF has shown 38% and 58% more improvement in recall than KBCS and IDCS, respectively. It has also retrieved relevant methods for all the nine features, whereas KBCS and IDCS have retrieved for seven and three features, respectively. This is because SBMF has checked feature-wise similarity among the methods and selected proper terms to construct the index. On the other hand, both KBCS and IDCS have used textual keywords in the methods as index terms without checking the appropriateness of the terms. The experiment was further extended to assess the retrieved codes quality in term of self-executability. All methods retrieved by SBMF have been self-executable because it has resolved data, method call, and library dependencies during index construction. On the other hand, KBCS and IDCS have retrieved, on an average 33.71% and 27.78% self-executable methods, respectively.

7.3 Threats to Validity

In this section, limitations of the experimental study are discussed in terms of internal, external, and construct validity.

Internal Validity In the experiment, there was no control over the skills of the subjects. However, the risks of this threat were reduced by applying repetitive measurement approach because same user created queries for KBCS, IDCS, and SBMF and evaluated the search results.

External Validity The set of features selected in the experiment, may not generalize to the population of software functions. However, these features are among the most common features used for the evaluation in code search [115].

Another possible threat is that, projects used in the experiment may not be sufficient enough. However, these projects are statistically representative of open source projects as highlighted in [7].

Construct Validity Existing code clone detection technique can be used to improve recall in code search. However, SBMF differs from code clone detection in several points. SBMF can detect similar methods written in different programming languages and only the execution of method is platform dependent. Another point is that code clone detection technique may provide false positive results to feature-wise clone detection (usually known as Type IV) if values of certain parameters are not defined properly [116]. As a result, search engine may retrieve irrelevant methods. However, SBMF checks dynamic behavior through executing method and matches the output for corresponding input to detect feature-wise similar methods. Such mechanism ensures that methods providing the same output, are feature-wise similar and thus no irrelevant method is added to these methods.

7.4 Future Work

This research contributes in the literature by devising a technique named SBMF. The technique increases the effectiveness in code search by indexing functionally similar methods under relevant terms. However, there are some scopes to improve the technique and these are addressed as follows.

- In order to determine the feature-wise similarity among a set of methods, SBMF executes those methods against a set of input data. It is computationally expensive to execute against a large input dataset. The future work of this research is to minimize the size of the input dataset. The minimized dataset should cover all the behaviors of the methods and it will help to identify functionally similar methods.

- SBMF is proposed to retrieve relevant methods from the code base. Since developers are also interested to find source codes of the desired components. So, the technique will be further extended to provide component level searching facility in future.

- Current implementation of SBMF works for Java projects only. However, the proposed technique is language and platform independent. Only the fact extraction is language dependent. So, there is a plan in future to upgrade the prototype for supporting other languages such as C#, python, C, C++, etc.

- In order to evaluate the proposed technique, only open source projects were used in the experiment. In future, the technique will be evaluated on the industrial projects to observe its behavior.

Although the purpose of SBMF is to retrieve more relevant code fragments based on the functional similarities, some parts of the technique can be used in other research domains. For example,

- SBMF detects feature-wise similarities among a set of methods. The detection scheme can be used to find Type-IV code clones in a project.
- Functional similarity detection approach in SBMF can be used to detect two types of code smell such as *Duplicate Method* and *Oddball Solution*.
- Dependency resolution process of SBMF can be employed to generate method level program slice. It will help to find minimum dependencies to convert a method into a program slice.
- SBMF generates self-executable methods and detects all possible behavior of a method by generating an input set covering all branches. The technique can be employed to speed up the Test Driven Development (TDD) process. The reason is that developers do not write unit test cases unless there exists some extreme test cases. All the retrieved methods have already been tested by SBMF while constructing the index.

Appendix A

List of Stop Words

"a", "a's", "able", "about", "above", "according", "accordingly", "across", "actually", "after", "afterwards", "again", "against", "ain't", "all", "allow", "allows", "almost", "alone", "along", "already", "also", "although", "always", "am", "among", "amongst", "an", "and", "another", "any", "anybody", "anyhow", "anyone", "anything", "anyway", "anyways", "anywhere", "apart", "appear", "appreciate", "appropriate", "are", "aren't", "around", "as", "aside", "ask", "asking", "associated", "at", "available", "away", "awfully", "b", "be", "became", "because", "become", "becomes", "becoming", "been", "before", "beforehand", "behind", "being", "believe", "below", "beside", "besides", "best", "better", "between", "beyond", "both", "brief", "but", "by", "c", "c'mon", "c's", "came", "can", "can't", "cannot", "cant", "cause", "causes", "certain", "certainly", "changes", "clearly", "co", "com", "come", "comes", "concerning", "consequently", "consider", "considering", "contain", "containing", "contains", "corresponding", "could", "couldn't", "course", "currently", "d", "definitely", "described", "despite", "did", "didn't", "different", "do", "does", "doesn't", "doing", "don't", "done", "down", "downwards", "during", "e", "each", "edu", "eg", "eight", "either", "else", "elsewhere", "enough", "entirely", "especially", "et", "etc", "even", "ever", "every", "everybody", "everyone", "everything", "everywhere", "ex", "exactly", "exam-

ple", "except", "f", "far", "few", "fifth", "first", "five", "followed", "following", "follows", "for", "former", "formerly", "forth", "four", "from", "further", "furthermore", "g", "get", "gets", "getting", "given", "gives", "h", "had", "hadn't", "happens", "hardly", "has", "hasn't", "have", "haven't", "having", "he", "he's", "hello", "help", "hence", "her", "here", "here's", "hereafter", "hereby", "herein", "hereupon", "hers", "herself", "hi", "him", "himself", "his", "hither", "hopefully", "how", "howbeit", "however", "i", "i'd", "i'll", "i'm", "i've", "ie", "if", "ignored", "immediate", "in", "inasmuch", "inc", "indeed", "indicate", "indicated", "indicates", "inner", "insofar", "instead", "into", "inward", "is", "isn't", "it", "it'd", "it'll", "it's", "its", "itself", "j", "just", "k", "keep", "keeps", "kept", "know", "knows", "known", "l", "last", "lately", "later", "latter", "latterly", "least", "less", "lest", "let", "let's", "like", "liked", "likely", "little", "look", "looking", "looks", "ltd", "m", "mainly", "many", "may", "maybe", "me", "mean", "meanwhile", "merely", "might", "more", "moreover", "most", "mostly", "much", "must", "my", "myself", "n", "name", "namely", "nd", "near", "nearly", "necessary", "need", "needs", "neither", "never", "nevertheless", "new", "next", "nine", "no", "nobody", "non", "none", "noone", "nor", "normally", "not", "nothing", "novel", "now", "nowhere", "o", "obviously", "of", "on", "often", "oh", "ok", "okay", "old", "on", "once", "one", "ones", "only", "onto", "or", "other", "others", "otherwise", "ought", "our", "ours", "ourselves", "out", "outside", "over", "overall", "own", "p", "particular", "particularly", "per", "perhaps", "placed", "please", "plus", "possible", "presumably", "probably", "provides", "q", "que", "quite", "qv", "rather", "rd", "re", "really", "reasonably", "regarding", "regardless", "regards", "relatively", "respectively", "right", "s", "said", "same", "saw", "say", "saying", "says", "second", "secondly", "see", "seeing", "seem", "seemed", "seeming", "seems", "seen", "self", "selves", "sensible", "sent", "serious", "seriously", "seven", "several", "shall", "she", "should", "shouldn't", "since", "six", "so", "some", "somebody", "somehow", "someone", "something", "sometime",

“sometimes”, “somewhat”, “somewhere”, “soon”, “sorry”, “specified”, “specify”, “specifying”, “still”, “sub”, “such”, “sup”, “sure”, “t”, “t’s”, “take”, “taken”, “tell”, “tends”, “th”, “than”, “thank”, “thanks”, “thanx”, “that”, “that’s”, “thats”, “the”, “their”, “theirs”, “them”, “themselves”, “then”, “thence”, “there”, “there’s”, “thereafter”, “thereby”, “therefore”, “therein”, “theres, ‘thereupon’, ‘these’, ‘they’, ‘they’d’, ‘they’ll’, ‘they’re’, ‘they’ve’, ‘think’, ‘third’, ‘this’, ‘thorough’, ‘thoroughly’, ‘those’, ‘though’, ‘three’, ‘through’, ‘throughout’, ‘thru’, ‘thus’, ‘to’, ‘together’, ‘too’, ‘took’, ‘toward’, ‘towards’, ‘tried’, ‘tries’, ‘truly’, ‘try’, ‘trying’, ‘twice’, ‘two’, ‘u’, ‘un’, ‘under’, ‘unfortunately’, ‘unless’, ‘unlikely’, ‘until’, ‘unto’, ‘up’, ‘upon’, ‘us’, ‘use’, ‘used’, ‘useful’, ‘uses’, ‘using’, ‘usually’, ‘uucp’, ‘v’, ‘value’, ‘various’, ‘very’, ‘via’, ‘viz’, ‘vs’, ‘w’, ‘want’, ‘wants’, ‘was’, ‘wasn’t’, ‘way’, ‘we’, ‘we’d’, ‘we’ll’, ‘we’re’, ‘we’ve’, ‘welcome’, ‘well’, ‘went’, ‘were’, ‘weren’t’, ‘what’, ‘what’s’, ‘whatever’, ‘when’, ‘whence’, ‘whenever’, ‘where’, ‘where’s’, ‘whereafter’, ‘whereas’, ‘whereby’, ‘wherein’, ‘whereupon’, ‘wherever’, ”whether”, ”which”, ”while”, ‘whither’, ‘who’, ‘who’s’, ‘whoever’, ‘whole’, ‘whom’, ‘whose’, ‘why’, ‘will’, ‘willing’, ‘wish’, ‘with’, ‘within’, ‘without’, ‘won’t’, ‘wonder’, ‘would’, ‘would’, ‘wouldn’t’, ‘x’, ‘y’, ‘yes’, ‘yet’, ‘you’, ‘you’d’, ‘you’ll’, ‘you’re’, ‘you’ve’, ‘your’, ‘yours’, ‘yourself’, ‘yourselves’, ‘z’, ‘zero’, ‘quot’, ”

Appendix B

Language Specific Stop Words

“public”, “static”, “void”, “abstract”, “continue”, “for”, “new”, “switch”, “assert”, “default”, “goto”, “package”, “synchronized”, “boolean”, “do”, “if”, “private”, “this”, “break”, “double”, “implements”, “protected”, “throw”, “byte”, “else”, “import”, “public”, “throws”, “case”, “enum”, “instanceof”, “return”, “transient”, “catch”, “extends”, “int”, “short”, “try”, “char”, “final”, “interface”, “static”, “void”, “class”, “finally”, “long”, “strictfp”, “volatile”, “const”, “ oat”, “native”, “super”, “while”, “org”, “eclipse”, “swt”, “string”, “main”, “args”, “null”, “this”, “extends”, “true”, “out”, “println”, “try”, “catch”, “e”, “exception”, “stack”, “trace”, “Scanner”, “in”, “int”, “double”

Appendix C

Experimental Projects

Table C.1: Experimental Projects

Project Name	**LoC**	**Number of Source Files**	**Number of Methods**	**Size (in Byte)**
jgaap	2481	36	2800	3466562
water-simulator	12420	97	21996	2801608
imsmart	3000	44	4226	49706204
dsachat	7796	63	12246	988707
jdbacl	45419	268	365709	13266148
omjstate	2202	31	1621	6257242
beanbin	11361	201	88029	13410271
templatedetails	1257	3	352	5112426
inspirento	13448	103	41255	3107432
jsecurity	50357	479	700987	14951444
jmca	19807	58	21526	807700
tullibee	8174	40	12297	999546

Continued on next page

Project Name	LoC	Number of Source Files	Number of Methods	Size (in Byte)
nekomud	1526	22	818	7079173
geo-google	30105	115	120660	3700472
byuic	10020	24	3663	1341501
jwbf	19235	166	82075	8232557
saxpath	7813	29	10440	2587663
jni-inchi	8889	56	13808	13138596
jipa	721	5	90	228380
gangup	27818	188	123477	9103395
greencow	43	2	5	206633
apbsmem	13634	94	28227	6144891
a4j	9426	67	29670	994435
bpmail	4331	63	11783	7065727
xisemele	14406	114	34625	3228100
httpanalyzer	5872	37	2786	3898800
javaviewcontrol	8457	31	3894	2831243
sbmlreader2	1394	12	253	70503254
corina	87448	480	888138	19134189
schemaspy	17679	95	42641	1023394
petsoar	9649	124	36730	9237191
javabullboard	25861	140	127541	12839984
diffi	1593	20	1180	268585
gaj	982	23	1378	238779
glengineer	8115	56	15683	488098

Continued on next page

Project Name	LoC	Number of Source Files	Number of Methods	Size (in Byte)
follow	9751	90	25143	600826
asphodel	2220	38	2970	2160703
lilith	79142	492	1184354	15471293
summa	183366	1017	3897714	101810796
lotus	3187	108	12311	334436
nutzenportfolio	21869	131	102015	13667943
dvd-homevideo	4623	18	791	473202
resources4j	2561	19	1337	319861
diebierse	3212	34	3715	1198840
rif	3054	33	2650	10700600
biff	2532	8	124	277871
jiprof	37286	215	161429	2118108
lagoon	23584	157	76699	6782769
shp2kml	597	8	192	12355056
db-everywhere	18458	179	112766	11546424
lavalamp	6117	113	27522	5546075
jhandballmoves	12775	123	41533	2875239
hft-bomberman	24165	274	203754	19132920
fps370	3290	20	1117	4604858
mygrid	8852	71	26676	3218400
templateit	4543	36	4715	5227895
sugar	8106	62	12951	1705995
noen	70860	897	1924170	71122806

Continued on next page

Project Name	LoC	Number of Source Files	Number of Methods	Size (in Byte)
dom4j	70065	339	877538	11273321
objectexplorer	17532	158	89941	1695203
jtailgui	7224	91	14902	6386186
gsftp	4450	31	3130	700459
openjms	203373	1296	6091314	16779220
gae-app-manager	1260	18	711	2301962
biblestudy	5805	41	9245	585308
lhamacaw	56756	222	285945	2848590
jnfe	8278	126	31205	21221547
echodep	31446	160	68827	17140858
ext4j	5772	87	16826	6354303
battlecry	4122	22	1556	4303895
fim1	16860	120	60587	1590084
fixsuite	6937	48	10166	1114719
openhre	27602	214	264765	13742786
dash-framework	2158	54	2327	10012920
io-project	5467	46	5691	1263405
caloriecount	157464	1295	6541800	10109073
twfbplayer	22899	191	109336	2912391
sfmis	4357	48	10406	15673261
wheelwebtool	51328	263	454159	4830940
javathena	23596	105	114162	2637126

Continued on next page

Project Name	**LoC**	**Number of Source Files**	**Number of Methods**	**Size (in Byte)**
gaj	0	0	0	205830
ipcalculator	4657	20	1478	451552
xbus	52685	386	402548	22097822
ifx-framework	585013	7929	200915759	28970854
shop	8771	67	17532	1665754
at-robots2-j	28074	430	529777	3260646
jaw-br	7493	52	11453	8341046
jopenchart	10532	76	21605	763883
jiggler	54870	311	434955	4340031
gfarcegestionfa	8077	96	20386	4600615
dcparseargs	1080	13	621	256250
classviewer	3512	13	865	311292
jcvi-javacommon	169895	1428	8047465	14021629
quickserver	33053	224	226073	3783297
jclo	1147	8	365	258775
celwars2009	4184	18	1541	3111215
heal	42868	276	399508	10986210
feudalismgame	6114	63	12975	424338
trans-locator	751	10	270	256091
newzgrabber	9001	78	16450	476985
falselight	1011	16	350	582513

Appendix D

Experimental Queries

Table D.1: Experimental Queries

KBCS	**IDCS**	**SBMF**
sname_contents:((invertˆ10) AND (stringˆ10))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decodeˆ10) AND (stringˆ10))	sname_contents:((decodeˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decodeˆ10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((invertˆ10) AND (stringˆ10))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverseˆ10) AND (stringˆ10))	sname_contents:((reverseˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverseˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invertˆ10) AND (stringˆ10))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invertˆ10) AND (stringˆ10))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	**IDCS**	**SBMF**
sname_contents:((decode^10) AND (string^10))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invert^10) AND (string^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invert^10) AND (string^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invert^10) AND (string^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((rotate^10))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((invert^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((encrypted^10))	sname_contents:((encrypted^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypted^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encode^10) AND (url^10))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((hash^10) AND (md5^10))	sname_contents:((hash^10) AND (md5^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((hash^10) AND (md5^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((invertˆ10) AND (stringˆ10))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invertˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encodeˆ10) AND (stringˆ10) AND (urlˆ10))	sname_contents:((encodeˆ10) AND (urlˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encodeˆ10) AND (urlˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decodeˆ10) AND (stringˆ10) AND (urlˆ10))	sname_contents:((decodeˆ10) AND (urlˆ10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decodeˆ10) AND (urlˆ10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((filesˆ10) AND (filterˆ10))	sname_contents:((filesˆ10) AND (filterˆ10))return_sname_contents:((void))params_snames_contents:((File))	sname_contents:((filesˆ10) AND (filterˆ10))return_sname_contents:((void))params_snames_contents:((File))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((rotate^10))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))
sname_contents:((image^10) AND (resize^10))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((void))params_snames_contents:((String) AND (int) AND (int))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((void))params_snames_contents:((String) AND (int) AND (int))
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((invert^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((decode^10) AND (string^10))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((hash^10) AND (md5^10))	sname_contents:((hash^10) AND (md5^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((hash^10) AND (md5^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((camel^10) AND (case^10))	sname_contents:((camel^10) AND (case^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((camel^10) AND (case^10))return_sname_contents:((string))params_snames_contents:((string))
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((reverse^10))return_sname_contents:((string))params_snames_contents:((string))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String) AND (String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String) AND (String))
sname_contents:((invert^10))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((invert^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((camel^10) AND (case^10))	sname_contents:((camel^10) AND (case^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((camel^10) AND (case^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((encode^10) AND (url^10))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decode^10))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((encode^10) AND (url^10))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decode^10) AND (string^10))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((scale^10))	sname_contents:((scale^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (int))	sname_contents:((scale^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (int))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((encrypt^10))return_sname_contents:((String))params_snames_contents:((String))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((reverse^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((capture^10))	sname_contents:((capture^10))return_sname_contents:((Image))	sname_contents:((capture^10))return_sname_contents:((Image))
sname_contents:((decode^10))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((decode^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((unite^10))	sname_contents:((unite^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))	sname_contents:((unite^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))
sname_contents:((invert^10))	sname_contents:((invert^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((invert^10))return_sname_contents:((string))params_snames_contents:((string))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((decode^10))	sname_contents:((decode^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((decode^10))return_sname_contents:((string))params_snames_contents:((string))
sname_contents:((rotate^10))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))	sname_contents:((rotate^10))return_sname_contents:((void))params_snames_contents:((double))
sname_contents:((invert^10))	sname_contents:((invert^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((invert^10))return_sname_contents:((string))params_snames_contents:((string))
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((string))params_snames_contents:((string))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((string))params_snames_contents:((string))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((image^10) AND (resize^10))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((void))params_snames_contents:((int) AND (int))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((void))params_snames_contents:((int) AND (int))
sname_contents:((reverse^10) AND (string^10))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))	sname_contents:((reverse^10))return_sname_contents:((String))params_snames_contents:((String))
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((void))params_snames_contents:((string))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((void))params_snames_contents:((string))
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((void))params_snames_contents:((Image) AND (Int) AND (Int))	sname_contents:((resize^10))return_sname_contents:((void))params_snames_contents:((Image) AND (Int) AND (Int))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (double) AND (double))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (double) AND (double))
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image))
sname_contents:((decode^10) AND (url^10))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String) AND (char))	sname_contents:((decode^10) AND (url^10))return_sname_contents:((String))params_snames_contents:((String) AND (char))
sname_contents:((image^10) AND (resize^10))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((byte))params_snames_contents:((byte) AND (int) AND (int))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((byte))params_snames_contents:((byte) AND (int) AND (int))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (in))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (in))
sname_contents:((image^10) AND (resize^10))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((File))params_snames_contents:((File) AND (int) AND (int))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((File))params_snames_contents:((File) AND (int) AND (int))
sname_contents:((files^10) AND (list^10))	sname_contents:((files^10) AND (list^10))return_sname_contents:((List¡File¿))params_snames_contents:((File) AND (String))	sname_contents:((files^10) AND (list^10))return_sname_contents:((List¡File¿))params_snames_contents:((File) AND (String))
sname_contents:((join^10))	sname_contents:((join^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))	sname_contents:((join^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))

Continued on next page

KBCS	IDCS	SBMF
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (width) AND (height))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (width) AND (height))
sname_contents:((encrypt^10))	sname_contents:((encrypt^10))return_sname_contents:((byte))params_snames_contents:((byte))	sname_contents:((encrypt^10))return_sname_contents:((byte))params_snames_contents:((byte))
sname_contents:((join^10))	sname_contents:((join^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))	sname_contents:((join^10))return_sname_contents:((String))params_snames_contents:((List¡String¿))
sname_contents:((image^10) AND (resize^10))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((Image))params_snames_contents:((int) AND (boolean) AND (String))	sname_contents:((image^10) AND (resize^10))return_sname_contents:((Image))params_snames_contents:((int) AND (boolean) AND (String))

Continued on next page

KBCS	**IDCS**	**SBMF**
sname_contents:((image^10) AND (resize^10))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (int))	sname_contents:((resize^10))return_sname_contents:((Image))params_snames_contents:((Image) AND (int) AND (int))

Bibliography

[1] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, no. Apr, pp. 361–397, 2004.

[2] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[3] "krugle code search." `http://www.krugle.com/`. Online; accessed 20 November 2016.

[4] "merobase." `http://www.merobase.com/`. Online; accessed 17 May 2016.

[5] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE software*, vol. 25, no. 5, pp. 45–52, 2008.

[6] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *Proceedings of the 28th international conference on Software engineering*, (Shanghai, China), pp. 905–908, ACM, 2006.

[7] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?," in *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (Bremen, Germany), pp. 41–50, IEEE, 2015.

[8] W. B. Frakes and B. A. Nejmeh, "Software reuse through information retrieval," in *ACM SIGIR Forum*, vol. 21, pp. 30–36, ACM, 1986.

[9] S. K. Bajracharya and C. V. Lopes, "Mining search topics from a code search engine usage log.," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, (Vancouver, BC, Canada), pp. 111–120, Citeseer, 2009.

[10] C.-J. Lee, Y.-C. Lin, R.-C. Chen, and P.-J. Cheng, "Selecting effective terms for query formulation," in *Asia Information Retrieval Symposium*, pp. 168–180, Springer, 2009.

[11] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting

structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, (Portland, OR, USA), pp. 681–682, ACM, 2006.

[12] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: finding and leveraging implicit references in a web search interface for programmers," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*, (Newport, RI, USA), pp. 13–22, ACM, 2007.

[13] A. Begel, "Codifier: a programmer-centric search user interface," in *Proceedings of the workshop on human-computer interaction and information retrieval*, (Cambridge, Massachusetts, USA), pp. 23–24, 2007.

[14] S. E. Sim and R. E. Gallardo-Valencia, *Finding source code on the web for remix and reuse*. Springer, 2013.

[15] O. A. Lazzarini Lemos, S. K. Bajracharya, and J. Ossher, "Codegenie:: a tool for test-driven source code search," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, (Montreal, Canada), pp. 917–918, ACM, 2007.

[16] W. Janjic and C. Atkinson, "Leveraging software search and reuse with automated software adaptation," in *Proceedings of the 4th International Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, (Zurich, Switzerland), pp. 23–26, IEEE, 2012.

[17] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, (Vancouver, Canada), pp. 243–253, IEEE Computer Society, 2009.

[18] N. Sahavechaphan and K. Claypool, "Xsnippet: mining for sample code," *ACM Sigplan Notices*, vol. 41, no. 10, pp. 413–430, 2006.

[19] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, (Atlanta, GA, USA), pp. 461–464, ACM, 2007.

[20] O. A. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, (Hyderabad, India), pp. 212–221, ACM, 2014.

[21] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*, pp. 174–188, IBM Corp., 2010.

[22] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, 2006.

[23] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, (Boston, MA, USA), pp. 1589–1598, ACM, 2009.

[24] D. Spinellis and C. Szyperski, "How is open source affecting software development?," *IEEE Software*, vol. 21, no. 1, p. 28, 2004.

[25] J. Boegh, "Certifying software component attributes," *IEEE software*, vol. 23, no. 3, pp. 74–81, 2006.

[26] S. E. Sim, C. L. Clarke, and R. C. Holt, "Archetypal source code searches: A survey of software developers and maintainers," in *Proceedings of the 6th International Workshop on Program Comprehension (IWPC)*, (Ischia, Italy), pp. 180–187, IEEE, 1998.

[27] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[28] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[29] W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 617–630, 1994.

[30] M. Sojer and J. Henkel, "License risks from ad hoc reuse of code from the internet," *Communications of the ACM*, vol. 54, no. 12, pp. 74–81, 2011.

[31] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio, "Development with off-the-shelf components: 10 facts," *IEEE software*, vol. 26, no. 2, pp. 80–87, 2009.

[32] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, (Santa Fe, NM, USA), pp. 157–166, ACM, 2010.

[33] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[34] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[35] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Proceedings of the 15th Working Conference on Reverse Engineering*, (Antwerp, Belgium), pp. 155–164, IEEE, 2008.

[36] M. O'Keeffe and M. O. Cinnéide, "Search-based software maintenance," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, (Bari, Italy), pp. 10–pp, IEEE, 2006.

[37] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[38] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, (Santa Fe, NM, USA), pp. 167–176, ACM, 2010.

[39] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management Science*, vol. 54, no. 1, pp. 180–193, 2008.

[40] A. Mockus, "Large-scale code reuse in open source software," in *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development*, (Minneapolis, MN, USA), pp. 7–7, IEEE, 2007.

[41] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 180–193, 2010.

[42] C. Szyperski, J. Bosch, and W. Weck, "Component-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming*, (Lisbon, Portugal), pp. 184–192, Springer, 1999.

[43] G. T. Heineman and W. T. Councill, *Component-based software engineering.* Springer, 2001.

[44] S. E. Sim and R. E. Gallardo-Valencia, "Introduction: Remixing snippets and reusing components," in *Finding Source Code on the Web for Remix and Reuse*, pp. 1–14, Springer, 2013.

[45] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1, (Cape Town, South Africa), pp. 475–484, IEEE, 2010.

[46] F. S. Gysin, "Improved social trustability of code search results," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, (Cape Town, South Africa), pp. 513–514, ACM, 2010.

[47] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[48] R. C. Seacord, S. A. Hissam, and K. C. Wallnau, "Agora: A search engine for software components," *IEEE Internet computing*, vol. 2, no. 6, p. 62, 1998.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proceedings of the European Conference on Object-Oriented Programming*, (Kaiserslautern, Germany), pp. 406–431, Springer, 1993.

[50] R. P. Buse and W. Weimer, "Synthesizing api usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, (Zurich, Switzerland), pp. 782–792, IEEE Press, 2012.

[51] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, (Dubrovnik, Croatia), pp. 25–34, ACM, 2007.

[52] J. Singer, "Practices of software maintenance," in *Proceedings of the International Conference on Software Maintenance*, (Bethesda, Maryland), pp. 139–145, IEEE, 1998.

[53] R. E. Johnson, "Documenting frameworks using patterns," in *ACM Sigplan Notices*, vol. 27, pp. 63–76, ACM, 1992.

[54] L. Martie and A. van der Hoek, "Sameness: An experiment in code search," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, (Florence, Italy), pp. 76–87, IEEE Press, 2015.

[55] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st International Conference on Software Engineering*, (Garmisch Pattenkirchen, Germany), pp. 88–98, 1968.

[56] H. Schütze, "Introduction to information retrieval," in *Proceedings of the international communication of association for computing machinery conference*, 2008.

[57] C. Snijders, U. Matzat, and U.-D. Reips, "" big data": big gaps of knowledge in the field of internet science," *International Journal of Internet Science*, vol. 7, no. 1, pp. 1–5, 2012.

[58] P. Willett, "The porter stemming algorithm: then and now," *Program*, vol. 40, no. 3, pp. 219–223, 2006.

[59] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[60] J. B. Lovins, "Development of a stemming algorithm," 1968.

[61] C. Paice, "A word stemmer based on the lancaster stemming algorithm," in *ACM SIGIR*, pp. 56–61, 1990.

[62] G. Paltoglou and M. Thelwall, "A study of information retrieval weighting schemes for sentiment analysis," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, (Uppsala, Sweden), pp. 1386–1395, Association for Computational Linguistics, 2010.

[63] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[64] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination, "Apache lucene 4," in *Proceedings of the SIGIR 2012 workshop on open source information retrieval*, (Portland, Oregon, USA), p. 17, 2012.

[65] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[66] R. Prieto-Diaz and P. Freeman, "Classifying software for reusability," *IEEE software*, vol. 4, no. 1, p. 6, 1987.

[67] G. T. Leavens and M. Sitaraman, *Foundations of component-based systems*. Cambridge University Press, 2000.

[68] O. Nierstrasz and T. D. Meijler, "Research directions in software composition," *ACM Computing Surveys (CSUR)*, vol. 27, no. 2, pp. 262–264, 1995.

[69] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *ACM SIGPLAN Notices*, vol. 40, pp. 48–61, ACM, 2005.

[70] R. Holmes and R. J. Walker, "Supporting the investigation and planning of pragmatic reuse tasks," in *Proceedings of the 29th International Conference on Software Engineering*, (Hilton Minneapolis, MN, USA), pp. 447–457, IEEE, 2007.

[71] B. Dagenais and H. Ossher, "Automatically locating framework extension examples," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, (Atlanta, GA, USA), pp. 203–213, ACM, 2008.

[72] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, (Shanghai, China), pp. 54–57, ACM, 2006.

[73] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 195–202, IEEE, 2006.

[74] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, (Atlanta, GA, USA), pp. 204–213, ACM, 2007.

[75] Y. Ye, Y. Yamamoto, and K. Nakakoji, "A socio-technical framework for supporting programmers," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, (Dubrovnik, Croatia), pp. 351–360, ACM, 2007.

[76] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, (Atlanta, GA, USA), pp. 513–522, ACM, 2010.

[77] S. Bajracharya, J. Ossher, and C. Lopes, "Searching api usage examples in code repositories with sourcerer api search," in *Proceedings of the ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, (Cape Town, South Africa), pp. 5–8, ACM, 2010.

[78] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, (York, UK), pp. 385–400, Springer, 2009.

[79] "koders." `http://www.koders.com/`. Online; accessed 11 December 2016.

[80] "google code search." `https://code.google.com/`. Online; accessed 9 November 2016.

[81] O. Hummel and C. Atkinson, "Extreme harvesting: Test driven discovery and reuse of software components," in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, (Las Vegas, NV, USA), pp. 66–72, IEEE, 2004.

[82] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "Applying test-driven code search to the reuse of auxiliary functionality," in *Proceedings of the ACM symposium on Applied Computing*, (Honolulu, HI, USA), pp. 476–482, ACM, 2009.

[83] R. E. G. Valencia, *How Software Developers Solve Problems by Searching for Source Code on the Web: Studies on Judgments in Evaluation of Results and Information Use*. University of California, Irvine, 2012.

[84] Y. G. Gutierrez and L. A. M. Chile, "Kelluntekun: Rich internet application for collaborative snippet management, using oows2. 0 and vaadin," in *Proceedings of the 33rd International Conference of the Chilean Computer Science Society (SCCC)*, (Talca, Maule, Chile), pp. 107–115, IEEE, 2014.

[85] "sourceforge." `https://sourceforge.net/`. Online; accessed 9 November 2016.

[86] "github." `https://github.com/`. Online; accessed 17 May 2016.

[87] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proceedings of the 24th international conference on Software engineering*, (Orlando, FL, USA), pp. 513–523, ACM, 2002.

[88] F. McCarey, M. Ó Cinnéide, and N. Kushmerick, "Knowledge reuse for software reuse," *Web Intelligence and Agent Systems: An International Journal*, vol. 6, no. 1, pp. 59–81, 2008.

[89] R. Cottrell, R. J. Walker, and J. Denzinger, "Jigsaw: a tool for the small-scale reuse of source code," in *Companion of the 30th international conference on Software engineering*, (Leipzig, Germany), pp. 933–934, ACM, 2008.

[90] M. Goldman and R. C. Miller, "Codetrail: Connecting source code and web resources," *Journal of Visual Languages & Computing*, vol. 20, no. 4, pp. 223–235, 2009.

[91] D. Poshyvanyk, A. Marcus, and Y. Dong, "Jiriss-an eclipse plug-in for source code exploration," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, (Athens, Greece), pp. 252–255, IEEE, 2006.

[92] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, no. 1, pp. 349–414, 1998.

[93] T. Parr and R. Quong, "Antlr: A predicated," *SoftwarePractice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[94] T. Grainger, T. Potter, and Y. Seeley, *Solr in action*. Manning Cherry Hill, 2014.

[95] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE software*, vol. 11, no. 5, pp. 23–30, 1994.

[96] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Computing Surveys (CSUR)*, vol. 44, no. 1, pp. 1–56, 2012.

[97] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1317–1324, ACM, 2011.

[98] A. Alnusair, T. Zhao, and E. Bodden, "Effective api navigation and reuse," in *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, (Las Vegas, NV, USA), pp. 7–12, IEEE, 2010.

[99] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, (San Francisco, CA, USA), pp. 397–400, IEEE Press, 2013.

[100] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, (San Francisco, CA, USA), pp. 319–328, IEEE Press, 2013.

[101] B. Dagenais and H. Ossher, "Mismar: A new approach to developer documentation," in *Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), pp. 47–48, IEEE, 2007.

[102] A. Deshpande and D. Riehle, "The total growth of open source," in *Proceedings of the International Conference on Open Source Systems*, (Milan, Italy), pp. 197–209, Springer, 2008.

[103] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[104] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, (Portland, OR, USA), pp. 23–34, ACM, 2006.

[105] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, (Waikiki, Honolulu , HI, USA), pp. 111–120, IEEE, 2011.

[106] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[107] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, (Denver, CO, USA), pp. 34–45, IEEE, 2003.

[108] A. M. Zaremski and J. M. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 2, pp. 146–170, 1995.

[109] O. C. Zienkiewicz, R. L. Taylor, and R. L. Taylor, *The finite element method*, vol. 3. McGraw-hill London, 1977.

[110] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[111] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, (Hong Kong), p. 6, 2013.

[112] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[113] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proceedings of the 34th International Conference on Software Engineering*, (Zurich, Switzerland), pp. 178–188, IEEE Press, 2012.

[114] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011.

[115] O. A. L. Lemos, F. C. Ferrari, F. F. Silveira, and A. Garcia, "Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming," in *Proceedings of the 34th International Conference on Software Engineering*, (Zurich, Switzerland), pp. 529–539, IEEE Press, 2012.

[116] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "Xiao: tuning code clones at hands of engineers in practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, (Orlando, Florida, USA), pp. 369–378, ACM, 2012.