UNIVERSITY OF CALIFORNIA,
IRVINE

JCSSE: Java Code Snippet Search Engine

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCES

in Information and Computer Science

by

Phitchayaphong Tantikul

Thesis Committee:
Professor Susan E. Sim, Chair
Professor Donald J. Patterson
Professor Cristina V. Lopes

2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I wish to thank all those who have helped me in this dessertation. I would never have been able to finish the disseration without the guidance and support from my advisor, my friends and my family.

I owe my deepest gratitude to my advisor, Professor Susan Sim, who provided me invaluable guidance, encouragment, and demonstrated incredible patience entirely from the creation to the completion of this dissertation.

I would like to thank to my lab mates: Rosalva Gallardo for her continued assistance, encouragement and support in my writing; Albert Thompson for his innovative algorithm, thoughtful criticism and feedback; and Hye Jung Choi for her assistance and valuable opinions in this research and experiments.

I would also like to thank Sushil Barajchaya for sharing his enthusiasm for and comments on my work.

And finally, I would like to thank to my parents, my brother and friends for their love and support during years of my education.

ABSTRACT OF THE THESIS

JCSSE: Java Code Snippet Search Engine

By

Phitchayaphong Tantikul

Master of Sciences in Information and Computer Science

University of California, Irvine, 2011

Professor Susan E. Sim, Chair

Searching the web for source code has become a key activity during software development. It is common for programmers to look for a "snippet," that is a small piece of example code, to remind themselves of how to solve a problem or to quickly learn about a new resource. However, existing tools such as general-purpose search engines and code-specific search engines do not deal well with snippet search. In this thesis, we introduce the Java Code Snippet Search Engine (JCSSE) to fulfill this need. JCSSE is built on top of the Lucene text search engine. The repository is populated with over 122 000 code snippets that have been extracted from over 34 000 web pages from Java tutorial sites. The code snippets are parsed to make identifers searchable and the surrounding text is used as metadata. When presenting the search results, the snippets are accompanied by a short descriptor, which is the most relevant text paragraph as identified by topic modeling, also called Latent Dirichlet Allocation (LDA). This combination of searchable elements and features makes JCSSE robust and flexible.

Chapter 1

Introduction

1.1 Introduction

Searching for source code on the web has become an integral part of software development. We find evidence of this in the creation of search engines specifically designed to search for source code, such as Strathcona[12], Mica[20], Krugle[14], Koders[13], Google Code Search[9], and Sourcerer[6]. All of these tools take the approach of gathering together as much source code as possible from open source hosting sites and making the repository searchable. Unfortunately, this approach overlooks all the code snippets that are embedded in web pages that can be found throughout the Internet. Code snippets are valuable resources because they are small examples and are helpful for learning or remembering. In this thesis, we make better use of these resource by creating a Java Code Snippet Search Engine (JCSSE).

JCSSE is a web-based search engine for Java code snippet. It presents code snippets as search results along with some brief description extracted from the same tutorial page, so users can find a list of related code snippets and choose a code

snippet that is most suitable. Furthermore, after users choose a code snippet and are trying to understand it, users can also go to the tutorial page where the code snippet comes from, in order to get some background knowledge or contextual description which can help users to understand and adapt it to use faster.

The repository in JCSSE came from crawling 33 tutorial websites. These pages tend to contain two different types of information: code and natural language text. From these pages, we extracted code snippets from the surrounding textual content using Thompson's method [21]. The code snippet is treated as the primary document in the repository and the nearby text as metadata. Both types of input were tokenized and the code snippets were subsequently parsed as source code, prior to being indexed along with a web site description.

We provided both a basic and an advanced user interface. The basic user interface consisted of a simple text box, just like a typical search engine. The advanced interface allows users to search for a specific type of identifiers such as class names, method names, and imported packages for users who want to find an exact usage of a particular class or method.

We ranked the search results using combined relevance scores generated from both the code snippet and the metadata. We calculated the relevance (TF-IDF) separately for each part and then combined them.

When presenting the results, we provided a short summary that was taken from the natural language metadata. We conducted some empirical studies that showed that the best summary came from the paragraph immediately above a code snippet.

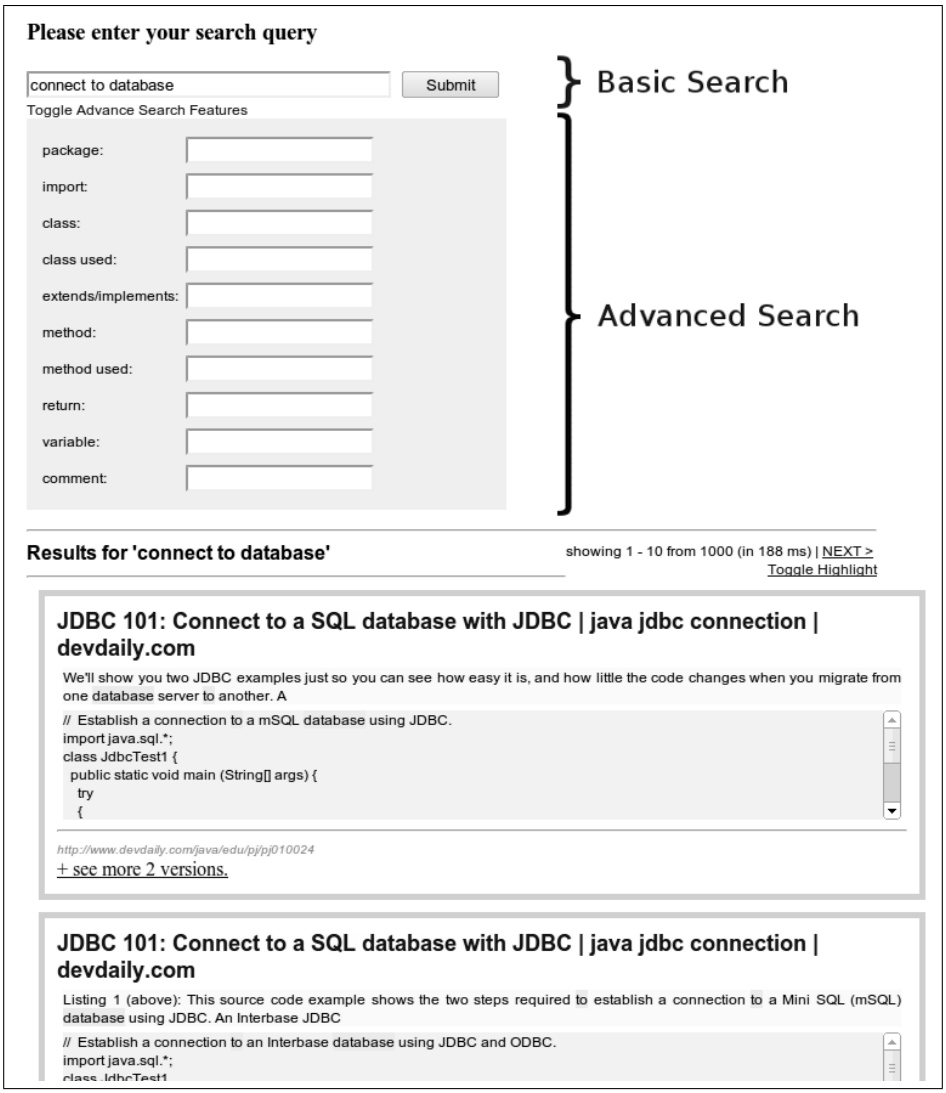


Figure 1.1: User interface of JCSSE

1.2 Organization

In this thesis, I present our observation on implementation process of a code snippet search engine, along with problems and solutions for them. This thesis has been separated into 8 chapters. Chapter 1 is introduction for this work, including motivation, and brief description of the approach. Next, chapter 3 explains related researches and related technologies to this problem. Chapter 5 describes architecture of the system and construction of code snippet repository. In chapter 6 further explains indexing

method of the search engine, and user interaction component. Search result ranking is presented in chapter 7 with result page design decision. Chapter 8 present my analysis on finding a short description for a code snippet. Lastly, chapter 9 concludes the thesis.

Chapter 2

Motivation

Searching for code on the web has become common place, due to the growth of the web and the diversification of content. Previously, source code was found primarily in repositories, such as SourceForge. But with the growth of the read-write web, it has become easier and easier for software developers to share information on the web, not only through static web pages, but also blogs, mailing lists, wikis, comments, and Twitter. Now, source code can be found on these sites as well. This spreading availability of source code has led to new use cases.

In this chapter, we review some of the key technological advances on the web that sets the stage for new kinds of code search, provide a motivating scenario for our work, and an overview of our tool, the Java Code Snippet Search Engine (JCSSE).

2.1 Growth of the World Wide Web

Over the last 15 years, the kinds of material available on the web has diversified and new technologies for sharing have appeared. Information can be found not solely

on authoritative web pages, but also in blogs, mailing lists, comments, wikis, and tweets. People are are not only reading content, they are also writing their own content. Producers make material available in the internet for the purpose of sharing their experiences, stating their opinions, and to communicate with others. This type of information sharing allows people to more easily find solutions to their specific problems.

The influences and trends have extended into software development communities. If a developer has a problem, they naturally turn to the web for information. By the same token, they are also likely publish solutions to problems that they have not seen elsewhere.There are many web pages that contain source code snippets. Examples include discussion forums, blog posts, tutorials, articles, and wikis. These data are available online in the internet for other software developers to read and use the information on the posts, and they can also contribute by answering the question or commenting on the post for better solutions.

Search engines are a key technology for locating the required information from the billions of available web pages. Everyone regularly uses general-purpose search engines, such as Google. As well, vertical search engines, have been created for specific domains. For instance, Koders and Krugle are code-specific search engines. Despite their capabilities, there are certain kinds of searches that they do not handle well.

2.2 Scenario

Consider the following hypothetical scenario.

Max, age 24, is a Java developer, who was hired a large company immediately after completing a degree in Computer Science. He is trying to find a way to connect

to a MySQL database that contains product and price information. He needs a solution in Java, so he can drop it into is code right away. He starts with connecting to the database, since it is the first step of his task. He has experience in connecting to MySQL before, but he doesn't remember the details. Therefore, he needs to search for an example to refresh his memory.

Figure 2.1: Comparing Search Result between Google and Google Code Search

Max starts by going to Google website to search for using keywords that describes his problem, such as "Java connect to MySQL database." When he looks at the search results, he finds no indication of which web pages contain source code. Therefore, he has to go to each webpage returned and scroll to find example code. The first page returned gives long background description of JDBC (Java Database Connector) and has example code at the bottom. But he misses the example, because it takes more time than he wanted to spend to skim through the entire page to make this discovery. Max looks at a second search result, hoping to find shorter page to read. This web page comes from a forum, and the example code is even harder to locate because it is buried in a comment.

Frustrated by the absence of example code, Max turns to a code-specific search engine, such as Google Code Search. This search engine allows him to zoom in on code from his programming language of choice by adding "lang:java" to his search.

He forms a query with that term and the keywords “connect to database”. The search results includes some descriptive information, such as a title or file name, followed by 3-6 lines of source code. Now every result contains source code. But it is difficult to figure out what the source code in each of the matches does. Many of them were returned because the search terms matched with comments or identifier names and they do not have what he needs. Furthermore, he needs to drill down into the file to see the context to make sense of code.

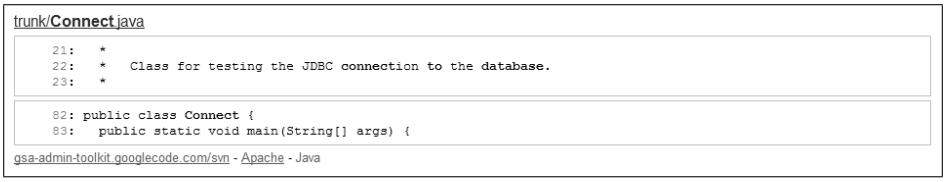


Figure 2.2: Example of a search result that does not contain working code in its snippet

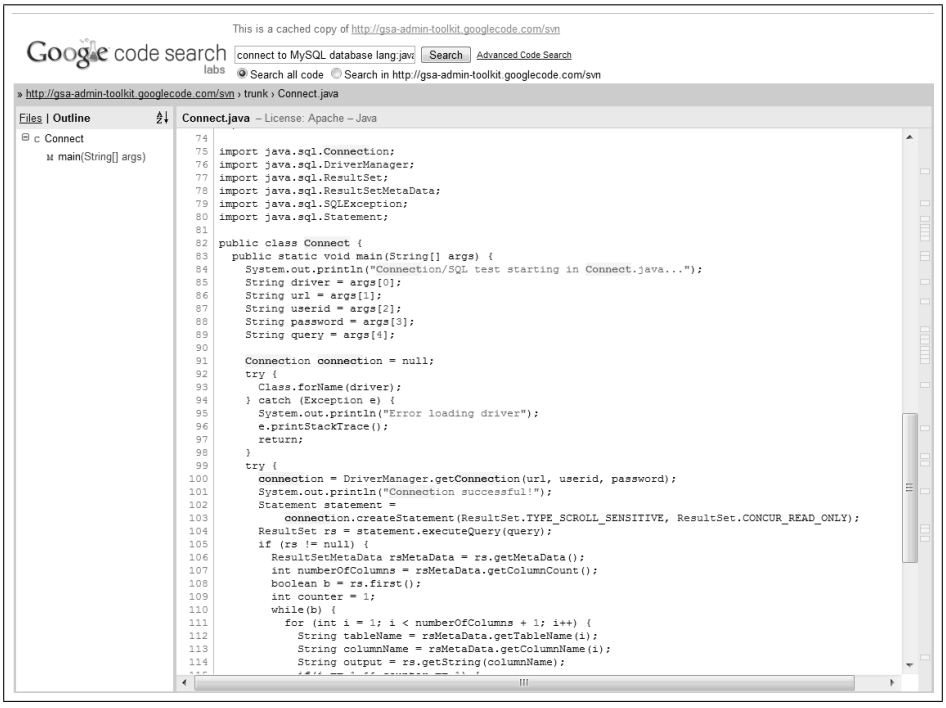


Figure 2.3: Example of a source code file that lacks comments

He randomly clicks on one of search results from the front page to examine the code. In the source code page, Max finally found some code that is starting to jar his

memory. Unfortunately, the code does not provide any explanation or documentation to help him along. He is limited to using code comments, but they are few and brief. There is also much more functionality than he needs, not just connecting to database. Therefore, locating the few lines of code that he needs is a difficult job.

2.3 Java Snippet Search Engine

Max needs a search engine that will allow him to search specifically for search snippets and display informative metadata along with the results. This is the problem that the Java Code Snippet Search Engine (JCSSE) is intended to solve. Figure 2.4 shows a screen shot of our tool. This tool seeks to address an important gap in the functionality provided by text-based and code-specific search engines.

Most search engines in use are text-based search engines. They performs search on textual data by simply using a matching algorithm to match text on webpages with user-specified keywords. This works well enough most of the time for most people. Usually, the entire page doesn't have to match, it only has to be close or relevant. As well, it is relatively easy to skim or summarize a page.

While text-based search engines are useful for typical searches, they have some shortcomings when it comes to source code. This mismatch comes from differences between the structure of text and the structure of source code. In other words, the grammar and syntax of the two languages are different. Most of the documents are simply written in natural language, which text-based search engines handle well. But these same algorithms fall short when they treat source code as text.

For example, one of popular search engine ranking method is called "Term Frequency/Inverted Document Frequency" (TF-IDF). It determines relevance based on

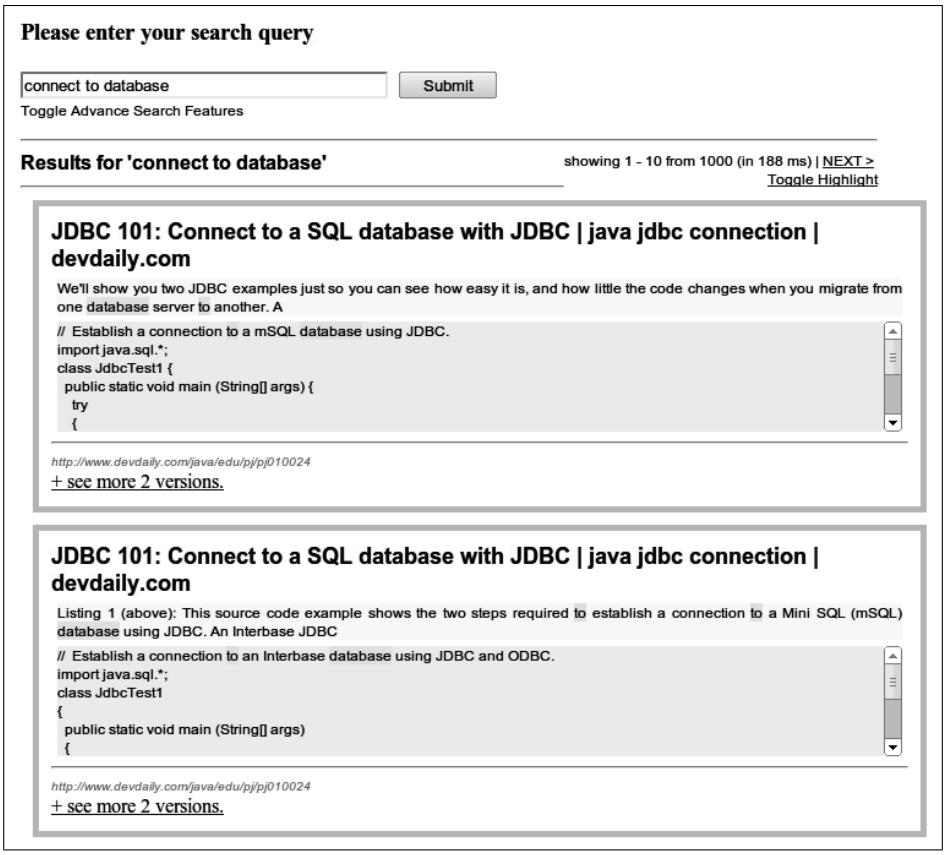


Figure 2.4: Example Search Result from JCSSE

the number of appearances of document keywords. If a text document contains the word "government" more frequently than usual, this document is likely to be relevant. Whereas in source code, most frequent words cannot be considered as an important word of the document. Words like "void", "int", "char" appear very often, while a class name, which is very meaningful, only appears once or twice in a source code file.

Consequently, many code-specific search engines were created to address this gap. These tools parse source code in from different programming languages and can search for specific types of data, such as method name or variable name. Moreover, code-specific search engines populate their repositories using data (source code files) collected from various open source repositories, such as SourceForge, FreshMeat, GoogleCode, GitHub.

	What is provided by these search engines		
What Max needs	**Google**	**Google Code Search**	**JCSSE**
Max needs code as returned result.	Web pages,may or may not contain code	Code from open source projects	Code snippets from web pages with description
Max needs code snippet displayed in search result	Maybe displayed	1-3 groups of 3 lines of matched snippet	full code snippet
Max needs explanatory text in search results	Yes, text snippet	No	Yes, text snippet
Max needs to filter for Java code only	Somewhat	Yes	Yes
Max needs to be able to search for specific element in code	No	Yes, - Regular expression - package - filename	Yes, - name of class - name of used class - name of method - name of used method - name of variables - package name - name of classes used in "extends" and "implement" - return type - comment
Max needs to be able use free text search	Yes, stemming and thesaurus	No, exact word search only	Yes, stemming keywords

Table 2.1: Characteristics of each search engine

Although these code-specific search engines can find source code for developers, they do not use the internet to its full potential. As stated before, the internet is now growing and source code does not appear only in open source projects. We can frequently see snippets are embedded in various web pages. Code-specific search engines do not yet take advantage of this data.

Table 2.1 summarizes the challenges and opportunities surrounding searches for code snippets. It shows how general text-based search engines, code-specific search engines, and JCSSE does or does not provide the features needed by Max in our motivating scenario.

Chapter 3

Related Work

In this chapter, we will present work related to ours in two areas. First, we will discuss the source code search engine tools that are currently available on the Web. Second, we will discuss the concepts of information retrieval that will be used to describe our approach to create a search engine for source code snippets.

3.1 Code Search Engines

There are many tools available on the Web that help users to search for source code. Some of these tools are commercial and others are built for research purposes. We will describe four tools that are related with our research: Koders[13], Sourcerer[6], Assieme[11], and Mica[20]. For each tool, we will describe its goal, characteristics of its repository, how results are shown, and some limitations.

3.1.1 Koders

Koders[13] is a commercial web-based source code search engine owned by Black Duck. Koders allows users to search for source code in 34 programming languages including Java, C++, C#, Perl, and Python. It allows users to search for source code files and open source projects.

Koders repository is built using crawlers to get open source projects and their information from over 4,500 web sites, including major open source repositories such as SourceForge.net, Cpan.org, RubyForge.net, and PlanetSourceCode.com. Koders' repository contains more than 1 billion of files or tens of billions of lines of source code. The repository is updated by crawling the same sites every 3-4 weeks to get new or updated source code. Overall, Koders is one of the code search engines that provides large and up-to-date source code in the Internet.

Figure 3.1 shows the results Koders returns when a user enters the keyword "hash map" in the search box. The search results present a list of files which contents match with the query. For each search result, a preview of its content and some related metadata is shown. The preview section displays a maximum of 5 contiguous lines that contain the query keywords. This preview helps users to get an idea of what is inside the file, and how the class/method/variable that users are searching for, are being used. The metadata section shows the language, license, copyright, number of lines of code, project name and path to the file. On the right hand side of the screen, the search engine also shows a list of projects that matches with the search query. This list would be useful for those who are searching for an open source project to use or reuse.

Koders shows some lines of source code for each search results. However, each result does not have any information on what the source code actually does. Users

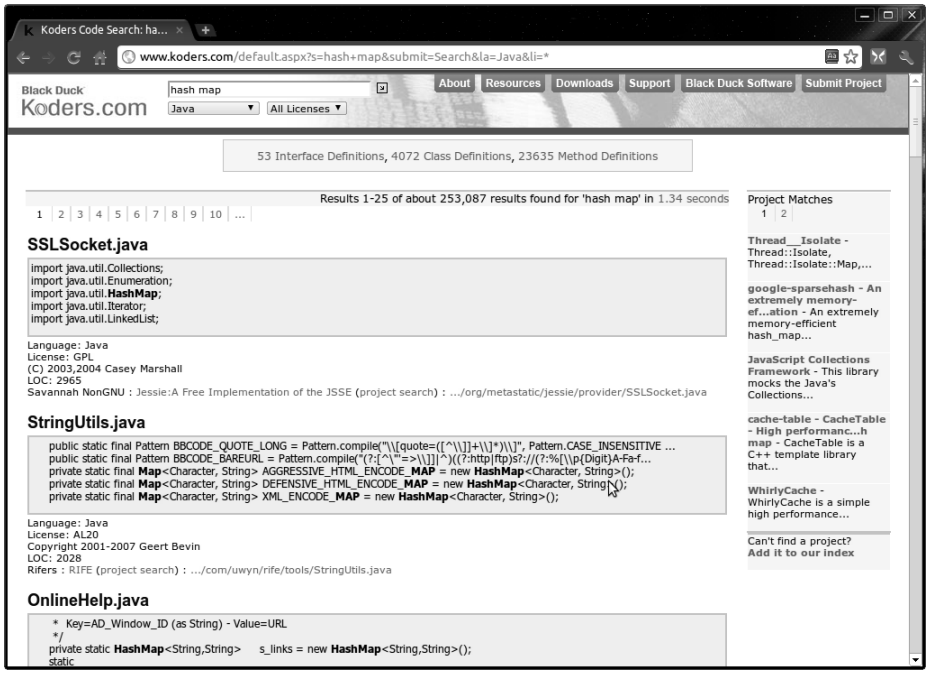


Figure 3.1: Searching for "hash map" in Koders.com

might get very little clue on the purpose of the file by only knowing the file name and seeing a few lines of source code. Users would need to read the file contents and understand it on their own. Users would benefit from having a short description of the source code that would help them decide which result is more appropriate for their needs.

3.1.2 Sourcerer

Sourcerer [6] is a web-based source code search engine that extracts fine-grained structural information from the code. It was developed at University of California, Irvine. It supports search for Java language source code. Sourcerer allows users to search by components, functions, fingerprints, and all of the previous ones. For each type of search, it allows the user to enter keywords to search for.

Sourcerer's repository contains Java projects from open source repositories, public web sites, and version control systems. It stores around 1,500 Java projects and 250,000 Java source files.

Figure 3.2 shows the results Sourcerer returns when a user enters the keyword "connect to database" in the search box. Each search results includes the name of the entity (ie: method, variable), the type of entity, the rank number, the uses of this entity, and the fingerprints. It offers the option to see the source inline (in the same web page), in full page, also browse the project, download the entire entity or just a slice. Sourcerer also shows information related to the project to which a piece of source code belongs to. It shows the name of the project, version, license, and category if available.

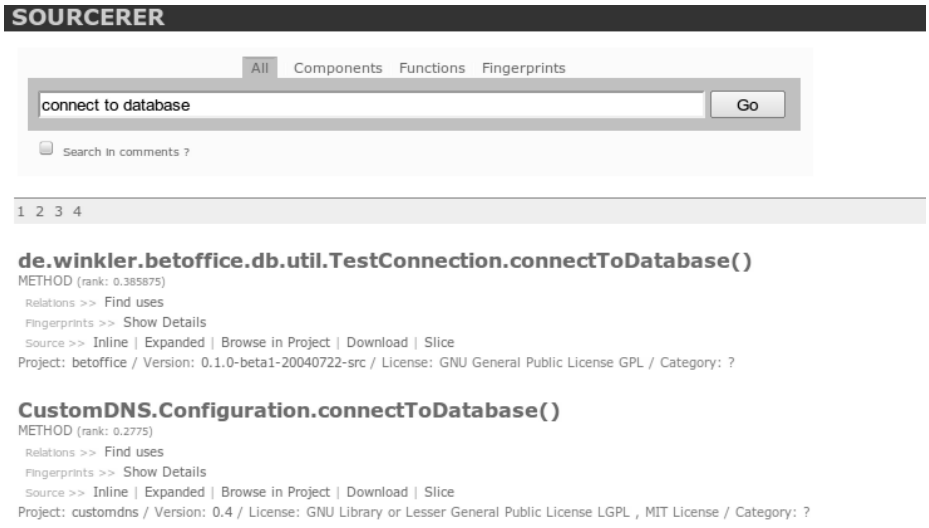


Figure 3.2: Example of searching in Sourcerer

The main purpose of the research behind Sourcerer is to find a representation of source code that can be used for searching and ranking by a source code search engine. A good representation of source code would facilitate users to find an appropriate search result faster.

Fields Available For Fingerprint Search In Sourcerer		
Control Structure	**Java Attributes**	**Micropatterns**
Synchronized	Modifiers	Designator
Starts	Interfaces	Pool
IF	Declared Constructors	COBOL Like
Instantiations	Fields	Immutable
MAX Loop Nesting	Declared Overloads	Box
Waits	Implements	Record
Joins	Field Self Type	Outline
SWITCH	Declared Methods	Pure Type
Path	Constructors	Implementor
Notifys	Static Initializers	Taxonomy
Loops	Overloads	Function Pointer
Lines Of Code	Parents	Stateless
Average Loop Length	Classes	Restricted Creation
	Method	Compound Box
	Declared Fields	Data Manager
	Parameters	Trait
	Overrides	Augmentted Type
		Overrider
		Joiner
		Function Object
		Common State
		Sampler
		Canopy
		Sink
		State Machine
		Psuedoclass
		Extender

Table 3.1: Fields Available For Fingerprint Search In Sourcerer

Regarding the representation of source code for searching, Sourcerer introduces some elements based on the source code structure that can be used to match source code with user's queries. Sourcerer allows users to search using structural attributes such as component name, function name, and fingerprints. Source code fingerprints are attributes specificaly used it to represet a piece of source code. Fingerprints could be separated into 3 mains categories: Control Structure Prints, Java Types Prints, and Micro Pattern Prints. Control Structure Prints capture the type and occurences of control flows in the code, such as number of loops, or number of if or switch. Java Type Prints captures the occurrences of various types of identifiers in the code such as number of fields, methods, or classes in the code. Lastly, Micro Pattern Prints capture elements based on simple design patterns, such as pool, immutable, or stateless. Table 3.1 shows a list of the different options to search for fingerprints.

Regarding ranking, Sourcerer uses Lucene ranking algorithm [17] which mainly uses term frequency - inverse document frequency (TF - IDF). Sourcerer developers tried to use PageRank technique in code searching. PageRank [8] method starts by creating a weighted direct graph that represents the relationship between each

webpage. A webpage has 2 types of links - incoming links and outgoing links. Like in research publishing, significance of a research is determined by number of paper cites, the importance of a page or rank of a page is determined by the sum of values of all incoming links called "back links". Sourcere used java classes as web-pages, and class-references and method reference as page links. Unfortunately, the experimental result showed that the method does not have significant improvement of the ranking of source code searching, comparing to TF - IDF [19].

3.1.3 Assieme

Assieme[11] is a web-based search engine that mainly focuses on searching for APIs and API examples related to the user's query. This tool is useful if users want to know how to use an API by finding examples online.

Assieme's repository includes web pages with code examples, Javadocs, and jar files. The web pages with examples were collected doing a Google search for words that appear commonly in Java. Javadocs were collected by doing also a Google search but this time using the keyword: overview-tree.html. Finally, the Jar files were donwloaded from sun.com, apache.org, java.net, and sourceforge.net. To identify that a web page had source code or not, it iteratively runs an error-tolerant Java parser and makes the suggested corrections until the block of text becomes sufficiently large and the number of syntax errors becomes sufficiently small.

The user can enter a query using keywords. Assieme will find and show the list of fully qualify name (FQN) of APIs related to the query that are in the tool's repository. For each API, it shows the number of examples found in the data base and a link to the Javadoc. The user can select to see the examples for one API at a time. In that case, the tool will show for each example: the title of the original web page, the

url, and a page summary. This page summary shows Java types and the libraries (Jar files) used in code examples. If the user hover over a search result, code example previews will be shown on the right hand side of the screen.

Although Assieme produces code examples as search results, this tool heavily focuses on finding an API first. The effectiveness of the tool can be limited when the right API for a user is not included in Assieme's database or when a user is looking for information not related with APIs.

Another advantage of Assieme is that it shows a summary of the web page. However, this summary is not presented as natural language but instead as a list of Java types which could make difficult to understand the goal of the example.

3.1.4 Mica

Mica [20] augments standard web search results to help programmers find the right API class and methods given a description of the desired functionality. It also helps programmers find examples when they already know which methods to use.

Micas does not have a repository of source code files or APIs. Instead, for each search requested, it performs a search in Google using the Google Web API. It also uses the spellchecking functionally of Google.

Mica receive a keyword query, which it is used to perform a Google search. then, Mica identifies relevant methods and class names by loading and analyzing the result pages from Google, identifying the code related terms, using frequency-based heuristics to determine which names are likely to be the most relevant for programmers. After analysis, Mica displays relevant methods and class names of the search in a side bar on the left. On the right, web search results are shown categorized by their

content; a Java icon indicates which results contain source code.

This approach is useful for developers when they are looking for how to use an API, but could not give relevant results when users are looking for code snippets not related with APIs.

3.2 Information Retrieval

The main purpose of an information retrieval system is to retrieve a document that is the most related to what the user is looking for. The user should provide some information to the system (a set of keywords) in order to describe the characteristics of what the user expects to find, so the system could be able to retrieve it from a repository. After receiving the keywords, the process of retrieval can be broken down into steps as follows:

- User sends a query
- Information retrieval system uses the query to match documents within its data storage
- The system retrieved some amount of documents
- The system presents those documents to users in a proper way
- The user picks some documents from the list returned by the system

In order to have an information retrieval system like this, the system should have certain components, which are listed here :

- Storage/Repository component : storing documents.

- Indexing component : preparing documents in repository for easy and fast retrieval.
- Matching component : performing matching between user queries and documents.
- Ranking component : ordering retrieved documents for users.
- User interfacing component : receiving query from users and returning search results back to users.

Each of these components will be explained here.

3.2.1 Repositories

A repository is a place where all documents are stored. A repository is similar to a library but instead of storing a large collection of books, it stores a large collection of documents.

Repositories can be implemented using several approaches. First, it can be implemented in one place, to reduce costs of communication, but it is hard to expand. Second, it can be implemented in a distributed way which is best for expanding for larger volume of documents, although the communication cost would be compomised. Regarding the technology to store a repository, it could be using files, databases, or others. It is important to keep in mind that a repository needs to be accessed most of the time to retrieve data by users, so it would be beneficial in performance-wise if implementation of the repository is allowing users to access to a specific document very quickly.

Repositories are populated with a collection of documents. The types of the data

vary by the application of the system. For example, if the system is designed for searching for websites in the Internet, in this case, the collection of documents for repositories would be a collection of webpages in the Internet. Or if the system will be used for retrieving source files that have been created in a company, the collection of documents would be source code files in that company.

In case the documents that will be kept in the repository are from the Web, a special method is needed to gather all the information to be included in the repository. This method is called "Crawling" which we will explain in next section. In order to crawl, a program called "crawler" needs to be implemented to traverse into the Internet, grab URL links, and extract the needed information from the webpages.

3.2.2 Crawler

A crawler is a program used for traversing into the Internet for collecting data from pages. A crawler starts its process by receiving an input page URL, called "seed URL". The crawler then goes to the page, collects the HTML of that page, and parses the HTML in order to get all the URL links on that page. The crawler may filter the links' URL to select only links that have URL under the same domain of seed URL, before it puts all selected links into a queue. Finally, it fetches a link URL from that queue one at a time and repeat the process mentioned above again and again, until the queue is empty. (For more details, see [16, Manning et al. 2008]) During this process, the collected HTML can be extracted further for other specific purposes, or simply be stored in a repository for further use.

3.2.3 Indexing

An index is created to acelerate the process of finding relevant documents in the repository that match with the user query. If an index is not created, each document in the repository would need to be examined when a search is run. The design of the index will depend on the characteristics of the documents in the repository. In general the following factors need to be taken into account when designing an index: method to add information to the index, storage of data in the index, disk space needed for the index, speed of lookup, and maintenace over time.

3.2.4 Matching

Matching is the process of calculating a relevance score between the user query and a document in the repository. The matching process uses the index to find quickly relavant documents in the repository. The relevance score calculated during matching is used to rank the results returned to the user.

3.2.5 Ranking

In a typical search system, the result page usually includes more than one match. It could be difficult for the users to choose one of the results without any clue of the differences between them and what will suit them the most. Then, ranking the results based on their importance comes into play. The most basic ranking method is term frequency - inverse document frequency (TF - IDF). This method uses text in the document along with a search query to compute the ranking value. It counts the number of occurrences of each query term that are in the document (Term Frequency) and multiplies it by the number of documents where the search terms appear (Inverse

Document Frequency).

3.2.6 User Interface

The user interface of an information retrieval system will allow users to provide the system with information to do searches. The user interface will also show the results returned by the system. The design of the user interfaces varies depending of the documents stored in the repository. Commonly, users will input a query using keywords. However, the user can receive some hints from the system about how to form queries, for example, using regular expressions. Traditionally, the results of a search will be shown in a list of 10 results for each page of results. However, others approaches to show results like thumbnails or with graphics are also being used and explored.

Chapter 4

Architecture of JCSSE

In this chapter, we describe the high-level design of JCSSE and key implementation details.

4.1 Overview

The architecture of JCSSE is divided into two parts, a back end that works offline and a interactive front end. These parts are depicted in Figure 4.1 below.

The back end consists of a repository built on top of Apache Lucene. Our contributions consist of the techniques for populating the repository with code snippets, and for creating metadata and indexes. The front end provides a user interface to the repository through a web interface. Together, these two components work collaboratively to make it a web-based code-snippet search engine.

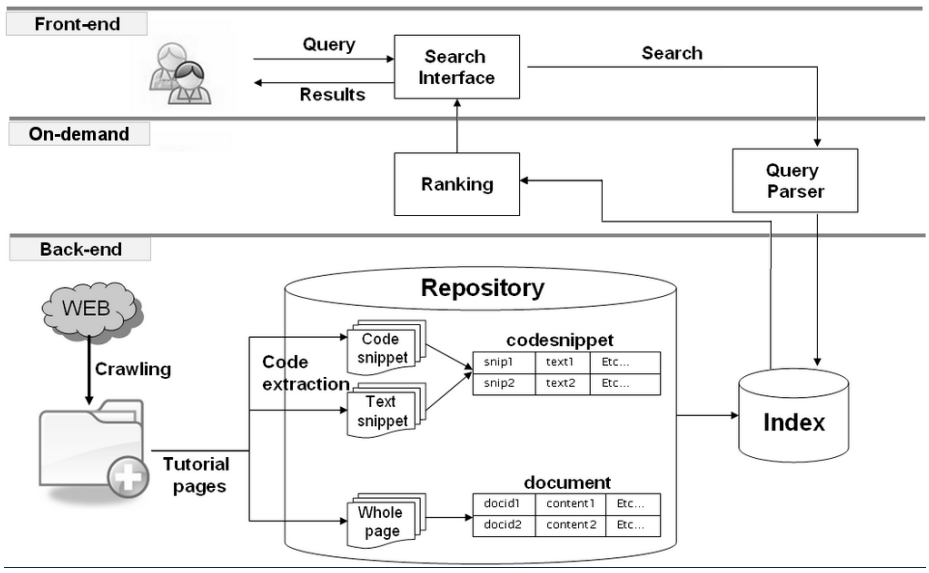


Figure 4.1: Overview Architecture

4.2 Back End

Our approach was informed by the following key insights:

1. Programming language tutorial pages contain an unusual combination of source code snippets and natural language.

2. The natural language on the pages can be used as metadata for the code snippets.

3. Effective searches for snippets need to make use of data in both the source code and the natural language text (metadata).

We implemented algorithms to crawl the web for Java tutorial pages, parse the pages to extract source code and metadata, and build indexes with this data. These steps will be described in detail in Chapter 5. All of these steps must be completed before user interaction occurs.

The repository was built using Apache Lucene, a text search engine library written in Java. It is an open source project that could be used to any application that needs full-text search. Some of its features include fast indexing, ranked searching, and many powerful query types: phrase queries, wildcard queries, proximity queries, range queries. It also allows to sort by any field and simultaneous update and searching.

4.3 Front End

The front end for JCSSE provides a user interface to the search engine. It is implemented as a web application using JSP. This component is web-based so users can put on queries and interacts with it easily, regardless of what operating system or what device the users are using.

We provide both a basic and an advanced interface. These are shown in Figure 1.1. The basic interface resembles a basic search engine interface. Users simply type their search terms into a box. With advanced interface, users can restrict search terms to certain parts of the code or metadata as will be shown in Table 6.1.

Chapter 5

Populating the Code Snippet Repository

In this chapter, we describe how the JCSSE was populated with data. We begin by giving a conceptual explanation of the algorithms that we developed for this purpose. We conclude by describing the contents of the repository in our prototype, as obtained by applying the algorithms.

5.1 Algorithms

We created a tool to harvest web pages and additional programs to process the pages in preparation for indexing, which will be described in the next chapter.

5.1.1 Crawling

The web crawler is a program to collect web-pages from the Internet. It takes a "seed URL" as input and places it in a queue. The program retrieves the page from the first URL in the queue and stores it locally. The contents of this page is further parsed for hyperlinks, which are added to the queue of web sites. The program then iterates through the remaining URLs in the queue. The crawler in this project was written from HTMLparser [2].

5.1.2 Pre-Processing

We pre-process the web pages to produce data to be indexed. There are two key steps in pre-processing: extraction of code snippets and text from web pages and filtering of irrelevant pages.

To extract code snippets, we use an algorithm developed by another student in our research group[21]. Given a web page, this algorithm will label each segment of HTML as a code snippet or a natural language text. The results are export as an XML file. Previous work showed that this algorithm labels code snippets in any HTML page with 95% correctness. Afer completion of this step, each web page has been factored into code snippets and text that can be used as metadata.

Next, we remove any web pages that do not contain any Java code. We do this in two operations, first by removing pages that do not contain any code snippets and then by removing pages that do not contain snippets in Java.

Pages that do not contain code snippets are not useful for the search engine and they need to be removed out of the repository. These are identified using a simple SQL query and deleted.

Since we are only interested in code snippets written in Java, we used some heuristics to identify whether a HTML page has any code snippet on it. We used common attributes of Java language for this filtering [10]. A page is considered to contain Java snippets if it has any of the following characteristics.

- Contains any keywords from Java
- Contains any programming syntactic symbols, e.g. =,+-.
- Contains identifiers from the Java.lang package

If a page does not have any of these symbols or terms, it was removed from the repository.

5.2 Contents of the Prototype Repository

To construct our prototype repository, we used a set of 33 seed URLs. These were chosen, because they were identified as rich sources of information, used a variety of page formats, and were used successfully in our previous research [21]. The full list of URLs are as follows.

1. http://java.sun.com/docs/books/tutorial/
2. http://learnola.com/
3. http://www.zetcode.com/
4. http://forum.codecall.net/java-tutorials
5. http://www.dickbaldwin.com/java/

6. http://www.learn-java-tutorial.com/

7. http://www.developer.com/java/

8. http://pages.cpsc.ucalgary.ca/~kremer/tutorials/Java/

9. http://www.beginner-tutorials.com/java-tutorials.php

10. http://www.javabeginner.com

11. http://www.javacoffeebreak.com/

12. http://www.cafeaulait.org/javatutorial.html

13. http://www.javaworld.com/

14. http://en.wikiversity.org/wiki/Java_Tutorial

15. http://leepoint.net/notes-java/index.html

16. http://www.javafaq.nu/java-example.html

17. http://www.java-tips.org

18. http://www.java2s.com/Tutorial/Java/CatalogJava.htm

19. http://www.java2s.com/Code/Java/CatalogJava.htm

20. http://www.java2s.com/Article/Java/CatalogJava.htm

21. http://www.java2s.com/Code/JavaAPI/CatalogJavaAPI.htm

22. http://www.java2s.com/Product/Java/GUI-Tools/CatalogGUI-Tools.htm

23. http://www.tech-recipes.com/category/computer-programming/java-programming/

24. http://www.exampledepot.com/egs/

25. http://www.devdaily.com/java/

26. http://www.roseindia.net/java/

27. http://en.wikibooks.org/wiki/Java_Programming/

28. http://www.codetoad.com/java/

29. http://danzig.jct.ac.il/java_class/

30. http://www.java-samples.com/showtitles.php?category=Java&start=1

31. http://www.algolist.net/Algorithms/

32. http://www.javapractices.com/

33. http://home.cogeco.ca/~ve3ll/jatutor0.htm

From these 33 seeds, 34,054 HTML pages were harvested. Applying the snippet extractor to these pages yielded 122,470 snippets. After pages that did not contain Java snippets were removed, 12,892 pages were left on the repository. These results are summarized in Table 5.1.

Total pages downloaded	34,054
After filtering pages with no code snippets	21,162
After filtering pages with no Java code	12,823

Table 5.1: Summary of number of pages after filtering

Chapter 6

Indexing

Indexing is the process to improve the search performance by ordering the data in an organized structure. We created our index using information from text associated with code snippets, code snippets, and web pages. In this chapter, we first explain how we associated a text segment in a web page with a code snippets. Then, we explain our parsing process to collect information to be included in our index from code snippets and text associated with code snippets. Finally, we present a summary of all the information we included as metadata for our index which is given to Lucene to do the indexing and matching for our system.

6.1 Pairing Text Segments and Code Snippets

Our main idea is that code snippets do not provide many keywords that are meaningful in natural-language because their text needs to comply with the syntax rules given by programming languages. Therefore, text in natural language surrounding code snippets should provide more meaningful keywords than code snippets them-

selves. The challenge of using text around code snippets is to decide which piece of text better describes a specific code snippet. To address this challenge, we did an empirical study to determine the location of a text segment that is more relevant to a code snippet. In this study, 200 pages from our repository were sampled based on a search using 4 keywords - binary tree, database, hashmap, socket. We sampled 50 pages for each keyword. The searching process was done using MySQL full-text search on full HTML pages with TF-IDF ranking score.

To validate our search results, we counted the number of pages that were relevant to the keywords. Out of 200 pages, 81 pages were identified to be related to the keywords. Figure 6.1 shows the number of relevant pages for each keyword. After some investigation, we found that some keywords are too common, such as database, hashmap, socket. Therefore, the words could appear in web pages that have a topic that is irrelevant to the search query.

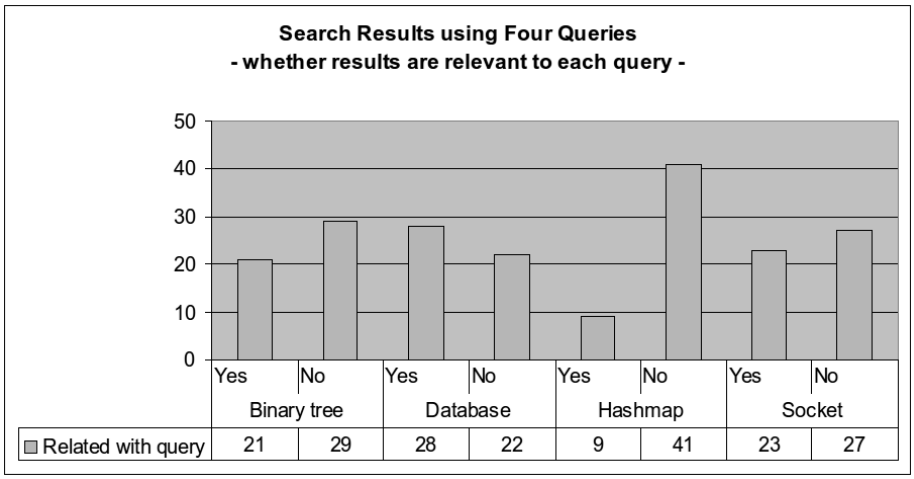
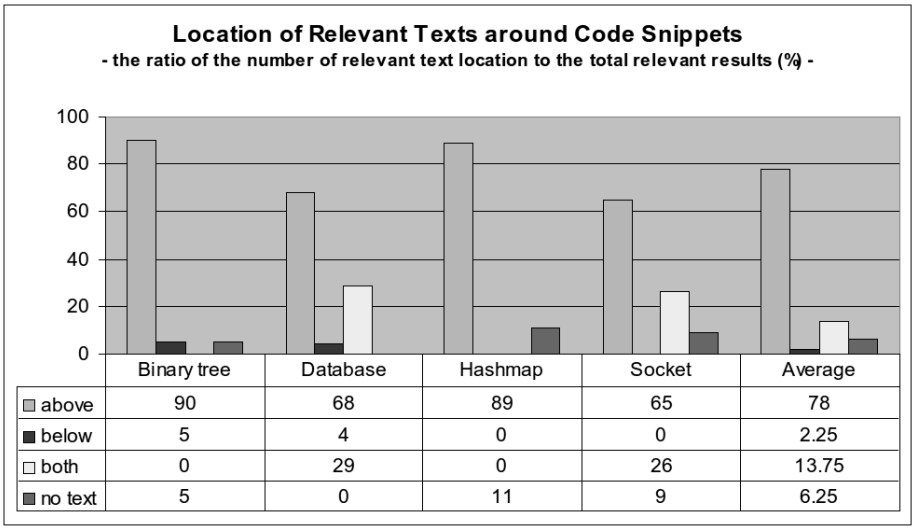


Figure 6.1: number of relevant result in each query

We used these 81 relevant pages in our study to find the location of a text segment that is relevant to a specific code snippet. In this experiment, code snippets were extracted and then these snippets were investigated manually to find the best text segment location for describing each code snippet. We found that an average of 78%

of code snippets has their best text description above them, an average of 2.25% in text bellow of them, and 13.75% in text both above and below of them. We could not find any relevant text description in the same page for 6.25% of code snippets. Figure 6.2 shows the results of our study.



	Binary tree	Database	Hashmap	Socket	Average
above	90	68	89	65	78
below	5	4	0	0	2.25
both	0	29	0	26	13.75
no text	5	0	11	9	6.25

Figure 6.2: Location of relevant text around code snippets

From this empirical study, we learned that the text located above code snippets best describes a specific code snippet. For this reason, we decided to pair the text above a code snippet with a code snippet together in the database. So that, words in the text segment can help describe a code snippet. We have a total of 43,306 pairs of text and code snippets in our repository.

6.2 Parsing

Commonly, after getting textual data, search engines will parse it into words and collect the words for matching with the users' query keywords. In this section, we explain how we parsed text segments and code snippets to extract information for our index.

6.2.1 Parsing Text Segments

Each text segment we collected will be parsed using simple word delimitors (e.g. white space, new line) in order to extract all words from the text segment. Due to the fact that many extracted words are very common and not very helpful for searching, such as: a, an, the, these words should be removed from the collection of extracted words. We use a list of stop words [3] to filter them out. The remaining words are changed to lower case and stemmed using the Porter Stemming Algorithm[18]. By ignoring letter cases and reducing each word to its simplest form, we increase the chances of words being matched with users' query keywords. All the keywords that were not filtered are included in a metadata field related to the code snippet.

6.2.2 Parsing Code Snippets

Not only text segments in our collection need to be parsed, but also the code snippets. Code snippets are different from the text segments because they can be viewed as structured textual data, while text segments can only be treated as unstructual data. One of the benefits of structural data is that it is easy to parse to get specific information from it. This benefit is used by other source code search engines such as [6][9][13]. These search engines allow users to search for a piece of source code indicating different identifier types, e.g. class name, variable name, and method name. Having structural information from a code snippet could be valuable for a source code search engine, since the search engine will be able to support users who want to search by specifying keywords and specific types of data. For this reason, we decided to parse and collect structural information from code snippets.

We collect 10 types of structural information from code snippets which are shown in 6.1. The first 9 types are mainly identifier declarations and invocations, which can

Extracted Identifier Types		
package	import	class declaration
class used	extending and implementing class	return type
variable declaration	method declaration	method invocation
comment		

Table 6.1: Extracted Indentifier Types from Parser

be generalized into terms of package, class, variable, and method information. These identifiers are common to be referenced to, and they are easy to be recognized.

The last piece of information to be parsed from code snippets is code comments. Search engines can benefit from comments because they provide information about the context of the code snippets and also contributes with some keywords that better describe code snippets. Having more keywords associated with a piece of source code will allow users to have more changes that a keyword included in a query matches with a keyword related to a piece fo source code. Therefore, code comments are collected and treated as textual information in our system. We are collecting javadoc comments, line comments, and block comments.

Parsing code snippets could be challenging. We cannot use a simple Java code compiler because many code snippets are not complete. It is common for an author of a web page to put only few lines or one method into a code snippet. Trying to compile a code snippet would produce a long list of syntactic errors. This issue could be possibly solved by building a new parser program based on Sun's Java syntax. However, this parser needs to support different types of input including a line of code, a block of code, a method, a class, or a complete source code file. This would make the algorithm for this parser very complicated. Thus, we decided to use an implemented library for the parsing process and try to adjust when a parsing error occurs. We used an abstract syntax tree parser (AST parser) from Eclipse library[15] to parse code snippets for identifiers and comments. This parser not only provides

a method to parse code, but it also provides 2 useful features; input type selection and error handling. The input type selection feature allows that a code snippet could be specified differently depending if it is a complete source code, a block of code, or a line of code. This allows the parser to produce more accurate output. Also, if we specified the type of a code snippet incorrectly, the parser will produce errors and we can change the input type for the code snippet for more precise parsing output.

Keywords from each of the 10 types of identifiers were put in a different metadata field each. Additionally, another metadata field was added to store all the keywords found in the code snippet. For this additional metadata field, we broke the camelcase identifiers, for the first 9 types of identifiers, into words to have more keywords associated with a code snippet. We did this because one characteristic of the identifiers is that they use "Camelcase" to join meaningful words together by using different character's case. In addition, for this metadata field, we also filtered the common words used in Java such as 'class,' 'for,' 'new,' and 'void.' We used the Sun's Java keyword list [10] to do this filtering.

6.3 Contents of the Index

Our index contains metadata from three different sources: web page, code snippet, and text as shown in 6.2. For web pages, we included two metadata fields: url and page title. For code snippets, we included 11 metadata fields: 10 for different identifier types and 1 for a summary of keywords found in a specific code snippet. For test, we included 1 metadata field that has the summary of keywords found in the text segment associated with a code snippet.

Information for all the metadata fields were indexed and stored in seperate columns

Web page	Code Snippet	Text
url page title	keywords from code snippet package import class declaration class used extending and implementing class return type method declaration method invocation variable declaration comments	keywords from text

Table 6.2: List of Indexable Metadata

in Lucene. We indexed words from 43,306 snippets which were compressed into indexes in Lucene with a total size around 71 MB.

Chapter 7

Matching and Ranking

Matching is the process of calculating a relevance score between the user query and a document in the repository. Ranking is the process of ordering the results to be shown to the user. In this chapter we explain how JCSSE handles both matching and ranking to show relevant code snippets to the users.

7.1 Matching

Matching the keywords in the users' queries with the keywords in code snippets should be an easy task since we already have all the keywords related to code snippets stored in indexes. However, we faced a challenging problem here: how to match the keywords in the users' query with our indexes which contain many types of metadata. Using many types of information could improve the performance of the search, but at the same time, it could cause a problem of information overload.

We found that using all the metadata fieds in our indexes at the same time to match with users' query brings many unrelevant results but reduces search time.

After experimenting with different combinations for the use of our indexes, we found that using three of our indexes had the most relevant results. These three metadata fields are: page title, keywords from code snippet, and keywords from text. Page title gives a general concept of the whole page. Keywords from code snippet contain identifier names that could explain what the code snippet does. Finally, keywords from text above code snippets, which proved to be relavant to the code snippet, could also contain words explaining the behavior of the code snippet. In other words, these three metadata fields provide a idea of the purpose of the code snippet.

We gave the user the possibility to use a general search or an advance search. For the general search, we only use the three metadata fields mentioned above and do not use the code specific metadata fields since they are already included in the keywords from code snippet field. Similarly, we do not use the URL field because it is less explanatory than the title of the page.

For the advanced search, the user can perform a search using special words such as "class:", "method:", or "variable:" placed in front of the query. These special words will explicitly tell the search engine to match the query keywords with words in the class name index, method name index, or variable name index accordingly. This advance search is useful when a user might know an identifier and want to match it with code snippets in the repository. Using our indexes, query keywords can be matched to any type of identifiers listed in table 6.1.

7.2 Ranking

We use the TF-IDF algorithm in Lucene to rank our results because it is the most common ranking algorithm utilized by many general search engines. Lucenes offers its

own version of scoring formula which is based on normalized TF-IDF algorithm.[17]

For the general search, we are using three indexes as was explained in the Matching section. Using these three indexes give us three different sets of search results. The challenge was to combine the results of these three sets because each set of results has different ranking scores and different results. We addressed this challenge by combining the three result sets using the snippet id. Basically, if any code snippet appears in all three result sets, we keep it, and calculate the new score by averaging its three ranking scores. By checking the code snippet's existence in all result sets, we guarantee its relevancy to the query. Furthermore, we tried different combinations of weight for calculating an average of the ranking scores. We found that giving equal weigth to each ranking scores yields to relevant results.

Chapter 8

Text Summarization for Search Results

In this chapter, we present the experiments and algorithms we used to provide the user with a relevant text that helps explain a code snippet in the search results. First, we explain what motivates us to summarize the text segment we already have related to a code snippet. Second, we explain how we framed this problem and how we plan to find a solution using the Latent Dirichlet Allocation(LDA) algorithm[7]. Finally, we show results from our experiment to determine the appropiate piece of the text segment to show to the user. We found that the paragraph that has the most matched topics with the code snippet is the best descriptor to show to the user in the search result.

8.1 Motivation to Summarize Text

We already have a text segment related to a code snippet in our repository. As explained previously in 6.1, we associated a code snippet with the text segment located above it on a web page. This text segment starts when the previous code snippets ends and includes all the text that is inmediately before a code snippet. The challenge we faced using this text segment is that it sometimes could be very long and not the whole text could be related to the code snippet. It would be useful to identify which part of the text segment is more relevant to the code snippet to be used to the user in the search results.

8.2 Summarization and LDA

The challenge we faced to identify relevant text in a text segment is similar to the problem of summarization. We need to summarize the content of the text segment based on their relevance to a code snippet. In other words, it is a problem of matching two set of concepts: the concepts of code snippets and the concepts of the text above the code snippet. How can we match these two sets of concepts? We could use word matching but the use of slang and synomyms could make the process complicated and not very accurate. It has been suggested [4, Asuncion et al.] that "Topic Modeling" could provide semantic information of a software artifact. Therefore, we decided to use topic modeling to address this challenge.

Topic modeling is a machine learning algorithm, its goal is to find a concept or topic of a text passage by analyzing all the words from all passage texts in the repository. This algorithm fits our problem very well because we could find topics from both code snippet and text segment and match them. In summary, using the

topic modeling algorithm, we can identify whether the selected text segment is related to a code snippet. Also, this algorithm can also help us to identify the correct size for the text description.

8.2.1 LDA Parameter Settings

Setting parameter to use LDA could be challenging. One of the difficult parameters to set is "Number of Topics", because the number of topics depends on all the data processed. The learning system cannot pre-determine the number of topics to use for the model until all data is processed. Therefore, it would be more difficult to set this parameter when dealing with dynamic input. However, in this project, our data is static.

Due to the fact that our data is static, we can calculate the number of topics that could best fit our topic model. We conducted an experiment with our data to simulate the learning process using different values for the "Number of Topics". We looked at the metric "Log Likelihood" that displays the fitness of the model to our data so that we can identify the value for the "Number of Topics" that best fits out data. In this experiment, we set the "number of topics" parameter to 50, 100, 200, 250, 300, 350, 400, 500, and 1000. We used Mahout [1] to learn topics from all HTML pages in our repository using those configurations. Results from this experiment are shown in Figure 8.1. This figure shows that the log likelihood score has its maximum value when the "number of topics" was set to 300. This means that the number of topics that best fits our data is 300. For that reason, we set that value for the "number of topics" parameter.

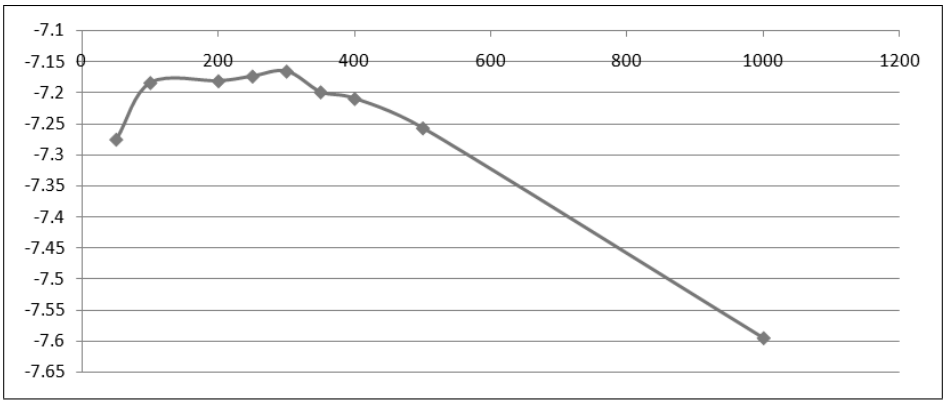


Figure 8.1: Log Likelihood score (Y-axis) on models with different "number of topics" (X-axis)

8.2.2 Determining Location of the Most Descriptive Piece of Text Segment

We conducted an experiment to determine which piece of the text segment helps explain a code snippet in the search results. We found that the paragraph that has the highest number of topics that matches with the code snippets give the best description which is both relevant and concise.

The data used in this experiment was taken by running 10 queries in our search engine. We took the first 20 results from each of these queries and we hand labeled them to train the algorithm. The keywords for the 10 queries used were ramdomly taken from a list of 555 common keywords found in an analysis of the Koders log [5].

To determine what part of the text segment is the most relevant for a code snippet, we evaluated the relevance of 4 pieces of information:

1. Text Segment: refers to a large text segment obtained from the code snippet extraction process. Text segments are textual data between two code snippets, mostly containing several paragraphs.

Query Keywords
1. smtp
2. quicksort
3. list
4. stringbuffer
5. date
6. webservice
7. signature
8. xpath
9. download file
10. base64

Table 8.1: Keywords Used for LDA Experiment

2. Last Paragraph: refers to the last paragraph of a text segment. In other words, it is the paragraph immediately above a code snippet.

3. Best Matched Paragraph: refers to a paragraph that has the highest frequency of matched topic keywords between the paragraph and its related code snippet

4. Page Title + Text Segment: refers to the title of the web page where the code snippet was found and also the text segment above the code snippet.This group provides the largest set of data related to a code snippet. We added these combination of information because it is used in the index.

In order to identify the Best Matched Paragraph we first obtained the words used in each paragraph. Then we used these words to find the corresponding topics for a paragraph. We also extracted the words from the code snippet to find the corresponding topics for a code snippet. Finally, we counted the number of topics that were the same between the topics in a code snippet and a paragraph. The paragraph that had the highest number of matches was the Best Matched Paragraph.

To calculate the percentage of matched topics for each of the 4 pieces of information on our data of 200 snippets for this experiment, we did the following for each

piece of information being evaluate:

- For each pair of code snippet and the text evaluated, we calculated two lists of topics: one for the code snippet and another for the topics for the text.

- Then, we calculated the number of matches between the two lists of topics. We consider that a pair is matched if it has one or more topics matched.

- Finally, we count the number of pairs that has matches and got the percentage of those from the total of 200 matches.

The results from this experiment are shown in Figure 8.2. In this figure, we see that both the Page Title + Text Segment and only the Text Segment have a high percentage of matched topics, with 83.16% and 80.54% respectively, which is expected since these two pieces of information had more content than the Best Matched and Last Paragraph. The Best Matched Paragraph has a 66.81% of matched topics and the Last Paragraph has a 59.26%.

Since our goal is to reduce the size of the text segment we decided to use the Best Matched Paragraph which is more concise than the Page Title + Text Segment and only Text Segment. And, at the same time the Best Matched Paragraph also provides more relevant information than the Last Paragraph.

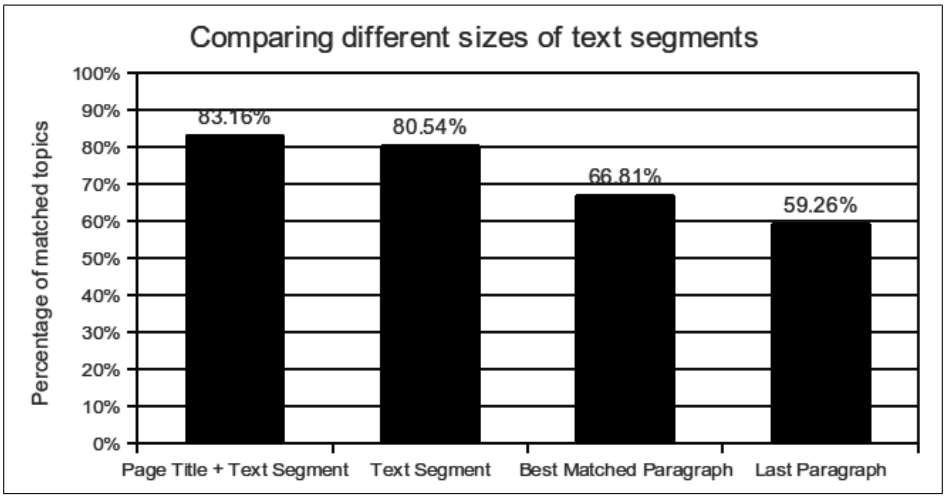


Figure 8.2: Comparison of Page Title + Text Segment, Text Segment, Most Matched Paragraph, and Last Paragraph

Chapter 9

Conclusion

In this thesis, we presented JCSSE, a search engine that capable of search for Java code snippets. This search engine is designed to help developers who are looking for a small chunk of source code to use as a reminder or to learn unfamiliar syntax. The JCSSE has been populated with over 34000 Java tutorial pages that have been crawled from the web. In the repository, the code snippets are treated as primary documents and the surrounding text treated as metadata. Users can search the repository using a basic or advanced interface, using both terms from the source code and the metadata. When presenting the results of a search, JCSSE provides a brief description for each code snippet in order to give its users more clues on what the code snippet could mean. By providing this information for each code snippet in search result, users could form better understanding of each code snippet and make better decision when picking them to incorporate with their project.

The contributions of this thesis consist of a proof of concept search engine for code snippets; novel algorithms for crawling, analyzing, matching, and summarizing code snippets, and insights into the problem of designing and building a snippet search

engine. Here are a few lessons learned from our experience.

9.1 Not every page from a programming tutorial website contains code

We expected to crawl web pages from code tutorial and example websites and to be able to use all of them in our initial repositories, but we found that not every one of them contains any source code. This was discovered after we crawled the web pages and were trying to extract code snippets from the pages. From 34 thousand web pages we have crawled, there are almost 13 thousand(37.86%) pages that contains no code snippet at all, in any programming language. And from that point, about a half of the rest of the pages is contains code snippets in Java programming language. Even we carefully chose seed websites to crawl to have a high chance of containing Java tutorials and examples, only about 13 thousand web pages (37.65%) of them have Java code snippets that can be served as a repository for our Java code snippet search engine.

9.2 Code snippets are often duplicated in the repository

After web pages were crawled and parsed into our search engine repository, we found many duplicate of code snippets in the repository. These came from both the same domain and different domains. It was not unusual for pages from one tutorial site to be copied wholesale, and possibly plagiarized, by another web site. As a result, the same snippet appeared multiple times in the search results. We dealt with this problem

by normalizing the formatting of the code snippets and hashing them. Snippets that mapped to the same bucket were collapsed and returned as a single match.

9.3 Text above code snippet is most related

When presenting a code snippet as a search result, we needed to provide a short description or text summary. In our investigation, we found that the text immediately above the snippet was the most closely related. In Chapter 6, we reported on an experiment to explore a relationship between code snippets and text segments from the same page. We discovered that an average of 78% of code snippets has a content that most corresponds to an immediate text segment above them. Therefore, we chose to pair between code snippet and its above text together, so we can use the text segment as a descriptor for the code snippet it pairs with.

The research in this thesis represents just a starting point for the work necessary to build a robust snippet search engine. Additional work is needed to improve the usability of the search engine, to enable the backend to work with other programming languages, and to incorporate other kinds of resources (such as emails and forums) in the repository. Finally, the effectiveness and helpfulness of a snippet search engine needs to be evaluated. Nevertheless, this thesis presents a proof of concept tool, some innovative algorithms for solving implementation problems, and insights into the design and construction of a snippet search engine.

Bibliography

[1] Apache mahout. `http://mahout.apache.org`, October 2010.

[2] Htmlparser. `http://htmlparser.sourceforge.net`, October 2010.

[3] stopwordslist. `http://www.ranks.nl/resources/stopwords.html`, October 2010.

[4] H. Asuncion, A. Asuncion, and R. Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 95–104. ACM, 2010.

[5] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 111–120. IEEE Computer Society, 2009.

[6] 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 *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, New York, NY, USA, 2006. ACM.

[7] D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine* 1. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[9] Google code search. `http://www.google.com/codesearch`, October 2010.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[11] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, New York, NY, USA, 2007. ACM.

[12] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. *SIGSOFT Softw. Eng. Notes*, 30(5):237–240, 2005.

[13] Koders. http://www.koders.com, October 2010.

[14] Krugle. http://www.krugle.com/, October 2010.

[15] T. Kuhn and O. Thomann. Abstract syntax tree. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, November 2006.

[16] C. Manning, P. Raghavan, H. Sch
"utze, and E. Corporation. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK, 2008.

[17] E. H. Michael McCandless and O. Gospodneti. *Lucene in Action*, pages 76–78. Manning Publications, second edition, 2010.

[18] M. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14(3):130–137, 1993.

[19] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval* 1. *Information processing & management*, 24(5):513–523, 1988.

[20] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. *Visual Languages - Human Centric Computing*, 0:195–202, 2006.

[21] C. A. Thompson. An evaluation of five algorithms to extract java code examples from web pages. Technical report, University of California, Irvine, 2010.