3-2016

A Search Engine for Finding and Reusing Architecturally Significant Code

Ibrahim Jameel Mujhid
ijm9654@rit.edu

A Search Engine for Finding and Reusing Architecturally Significant Code

by

Ibrahim Jameel Mujhid

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Software Engineering

Supervised by

Dr. Mehdi Mirakhorli

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

March 2016

The thesis ”Search Engine for Finding and Reusing Architecturally Significant Code” by Ibrahim Jameel Mujhid has been examined and approved by the following Examination Committee:

Dr. Mehdi Mirakhorli
Assistant Professor
Thesis Committee Chair

Dr. Meiyappan Nagappan
Assistant Professor

Dr. Stephanie Ludi
Professor

Dedication

To my beloved family and friends, for all of your support along the way

Acknowledgments

I am grateful to my thesis advisor Dr. Mehdi Mirakhorli. I could not have achieved this without his invaluable motivation, patience, and dedication to this work. I also thank Dr. Meiyappan Nagappan for serving on my thesis committee and his insightful comments and encouragement. Furthermore, I thank the rest of the department faculty and staff for their support through my study.

I would like to express my gratitude to Joanna Cecilia for her active participation in this research work and for her knowledge and contribution for it. Also, I would like to thank my team member Raghuram Gopalakrishnan for assisting in conducting experiments for my research. Finally, I would like to thank the Iraqi High Committee of Education Development for the full scholarship to complete my Master study.

Abstract

A Search Engine for Finding and Reusing Architecturally Significant Code

Ibrahim Jameel Mujhid

Supervising Professor: Dr. Mehdi Mirakhorli

Architectural tactics are the building blocks of software architecture. They describe solutions for addressing specific quality concerns, and are prevalent across many software systems. Once a decision is made to utilize a tactic, the developer must generate a concrete plan for implementing the tactic in the code. Unfortunately, this is a non-trivial task even for experienced developers. Developers often resort to using search engines, crowdsourcing websites, or discussion forums to find sample code snippets to implement a tactic. A fundamental problem of finding implementation for architectural patterns/tactics is the mismatch between the high-level intent reflected in the descriptions of these patterns ,and low-level implementation details of them. To reduce this mismatch, we created a novel Tactic Search Engine called ArchEngine (ARCHitecture search ENGINE). ArchEngine can replace this manual Internet-based search process and help developers to reuse proper architectural knowledge and accurately implement tactics and patterns from a wide range of open source systems. ArchEngine helps developers find implementation examples of tactic for a given technical context. It uses information retrieval and program analysis techniques to retrieve applications that implement these design concepts. Furthermore, the search engine lists the code snippets where the patterns/tactics are located. Our case study

with 21 professional software developers shows that ArchEngine is more effective than other search engines (e.g. SourceForge and Koders) in helping programmers to quickly find implementations of architectural tactics/patterns.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A complex software systems architecture is cautiously built to fulfill various concerns and needs such as security, usability, safety, dependability, and other vital qualities [7]. In order to fulfill and address such concerns, architects use architectural patterns, also known as styles, and architectural tactics to build the complete software system [23, 24]. The foundation of the entire software architecture is done using the architectural tactic, which comes in various shapes and sizes to tackle a range of quality issues. The usability of architectural tactics can be seen especially in fault tolerant and high performance systems. For example, reliability tactics such as redundancy with heartbeat, voting, and check pointing provides some solutions for mitigating, detecting, and recovery of faults, while performance tactics such as resource pooling and scheduling helps in optimizing response time and latency [10].

The implementation of such tactics in a robust and effective way becomes challenging for less experienced developers. This is because they contain variability points and huge number of design decisions that need to be taken care of before implementing the tactic. Any failure in implementing these tactics will lead to degradation of the complete architecture [34, 9, 39], therefore they may search on online forums and search engines because they did not understand how to implement specific tactics. Existing search engines and software repositories are the main sources that the programs use to learn or get ideas on how to implement a specific problem in their software or to reuse the existing source codes. However, this can be challenging when it comes to reusing fragments of architecture tactics due to the difficulties in identifying tactics in the source code [22].

To facilitate source code reusability problem, researchers have applied data-mining and natural language processing (NLP) techniques to build source code recommender systems [19], and search engines by matching keywords in queries to words in the descriptions of applications, comments in their source code, and the names of program variables and types. However, the primary intent of these techniques is retrieving generic functional code rather than tactical code. Difficulties in detecting patterns and lack of support in the current search engines are two main reasons as to why the notion of reusing architecturally significant source code is not well explored in the software architecture community.

In this work, we address these limitations by presenting a novel approach for automating the discovery, extraction, and indexing of architectural tactics across 116,609 opens source systems and to build a tactic search engine. Our approach utilizes advanced data mining techniques to reverse engineer the architectural tactics from a source code of 116,609 open source systems index, and use such knowledge to build ArchEngine. We built ArchEngine as part of a big data compatible architecture and conducted a case study with 21 professional programmers to evaluate this tactic search engine. The results show, with strong statistical significance, that users find more relevant tactical code snippets with higher precision using ArchEngine than those with other search engines such as Open Hub, Krugle, and GitHub.ArchEngine is available for public use at [1].

1.1 Motivation

Source code search is a fairly common task done by software developers. However, relatively little is known about how and why developers perform code search [33]. Not too many surveys have been done on this particular topic. A useful study has been published by Sim *et al* [37]. According to the study, code search is often done during the development and maintenance of software. They mentioned several reasons for code searching. The four main motivations have been mentioned as; (1) defect repair, (2) code reuse, (3) program

[1] `http://juno.main.ad.rit.edu:8081/ArchEngine/`

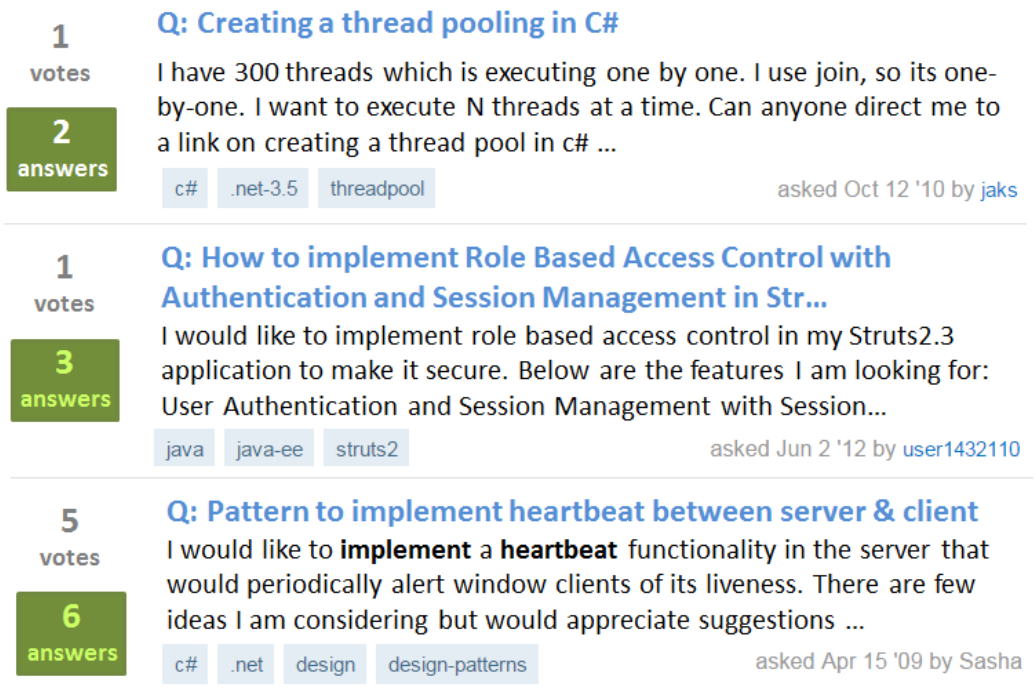


Figure 1.1: Developers seek help in online forums to implement architectural patterns/tactics

understanding, and (4) impact analysis.

With the increasing adoption of iterative incremental software development practices and integration of coding and design activities, there is an increasing need for a search engine that helps developers identify and reuse code snippets related to the architectural tactics. In a very trivial search, one can find several examples of developers posting requests for help to the online forums because they could not understand how to implement specific patterns/tactics. Figure 1.1 shows three examples of such questions. One developer is seeking help regarding the generic implementation of a Pooling tactic. While two others are looking for specific implementation of tactics in particular situations/technology. Another one developer wants to implement role-based access control along with Struts framework, while the third one is seeking samples to implement heartbeat reliability tactic between a client and a server. These examples show that typically developer's query for a sample tactical code has two parts, (i) the desired tactic and (ii) a particular context or technology

in which the tactic needs to be implemented. Therefore, a search engine not only needs to identify and index occurrence of architectural tactics, but also need to identify the technical context in which the tactic is implemented.

For example, a developer may be implementing a new feature in a software using *Heartbeat* tactic. She may look into the documentation and find out a method to accomplish a task, but the documentation may rarely have example code of how to use tactics. Therefore, she may search online for code that uses a Heartbeat. From the search result she may get the examples of the usage of the method. Existing code search engines e.g. *Google Code*, technical websites such as *StackOverflow*, and open source software repositories have been made to support different code search needs. They have been the primary resources that developers have used for reusing code, or even to obtain ideas to implement their software. However, this can be challenging when it comes to reusing fragments of architecture, design knowledge or what is so called architecturally significant code snippets [30]. A fundamental problem is related to the difficulties in identifying and tagging architectural patterns and tactics in the source code [22]. Therefore, current search engines fail to incorporate these design concepts in their underlying search algorithms. This may increase the percentage of non-relevant answers in search results. We, therefore propose a novel search technique that attempts to take into account the required architectural tactic as well as technical context (e.g. TCP, UDP, httpConnection, ...,etc) that are specified in the user query as input. By technical context we mean , framework, technology, programming languages, and/or APIs in which the tactic needs to be implemented. The search results returned by our system reflect those relationships among the technical problems and tactic. For example: if a user types "*Heartbeat for TCP*", then our system tries to find the best code snippets that implement Heartbeat for TCP connection.This is significant because this is first of its kind code search engine that enables developers to reuse tactical code snippets This novel technique makes our system different from any existing work.

1.2 Hypothesis

In this Thesis, we aim to evaluate the following hypothesis:

> ***Hypothesis****: ArchEngine users find more relevant architectural code snippets compared to Open Hub, Krugle, and GitHub users.*

Chapter 4 reports on the experiment conducted for this hypothesis. We aim to verify whether there is a statistically significant difference between the precision and accuracy of ArchEngine compared to other code search engines. We proposed a set of queries that a developer may come up with from a task assigned to 21 graduate students in architecture class. We give those queries as input to ArchEngine and other engines and save the output for each search engine. Consequently we manually scrutiny each of those results and find out whether they are accurate in our context or not, and compare these results with the results of Koders, GitHub,and Krugle. The results indicate a higher accuracy index of ArchEngine compared to other engines.

1.3 Challenges

There are several tasks associated with our code search system. We can , Roughly break down the major tasks into the following:

- **Collection**: The collection of large amounts of source code from open source repositories, in order to build a local repository for ArchEngine. The primary challenge in collecting source code from the online repositories is that there is no standard method of distribution. Open source projects are generally hosted by large open source repositories, such as SourceForge, GitHub, and Apache, which rarely provide hooks for performing this type of collection. The use of different version control systems, download protocols, and constantly changing format/content makes automating this collection process rather tedious.

- **Analysis**: We intend to do a statically analysis for projects in order to extract information and generate a dependency matrix for each source code file in the repository. In order to fully extract structural information from source code, the code must be declaratively complete; i.e. all the dependencies must be resolved. Unfortunately, this is become challenging when it come to source code from open source repositories for two reasons. First, diversity of programming languages used in the repositorys projects; it was hard to find a tool that could analyze all projects and generate dependencies matrix for them. Second, there is no guarantee that the code is declaratively complete; missing dependencies and incomplete files are quite common. Scaling the analysis to thousands of projects pose further complicates matters, as it eliminates the feasibility of performing any type of manual processing.

- **Searching for Tactical and Technical Context**: A direct approach for finding highly relevant code snippets is to search through the descriptions and source code of the projects to match keywords in queries to the names of the program variables and types. This approach assumes that programmers choose meaningful names when creating source code, which is often not the case. This is partially addressed by programmers who create meaningful descriptions of the applications in software repositories. However, it becomes challenging for developers to guess exact keywords because no single word can be used to perfectly describe a programming concept. The main challenge in this work is proposing an appropriate technique that enables the developer to search for tactics, as well as technical context through large scale repository in a light and scalable way to find highly relevant source code.

- **Prototype Development**: Due to the challenges posed by collection, analysis, and searching, it is often impossible to rapidly design and evaluate a useable application that enables developers to search through large quantities of source code .

- **Displaying Output**: Our search engine performs structural relationship search between tactical files and their neighbouring files. For the technical problem, the tool

searches the tactical file and its direct dependencies file to see whether the problem was implemented or not. Due to this nature of the search, the output display is different from the traditional full-text search engines. In full-text search engines, only the portions of a document which contain matched terms from keyword query are usually displayed. Structure based code search may require displaying code snippets that may be far apart in the same document or may consist of multiple documents. For example: A query ”Heartbeat for Tcp connection” may return a Doc1 that implement a method ”HeartbeatSender”. In Doc1, ”HeartbeatSender” may call a ”TcpConnection” method which is implemented in Doc2. In this case we have to show both Doc1 and Doc2. Displaying the output in a suitable manner is a daunting task and requires some mechanism to solve this. We have kept this problem out of the scope of this work.

Our main focus in this work is on item no. 3 mentioned above. In chapter 3, we discuss in details how we hope to address the challenges related to this item. For the challenges under remaining tasks, we do this work just to support the work under item no. 3.

1.4 Contribution of the Thesis

To the best of our knowledge, none of the aforementioned work exactly matches our approach. In this thesis, we make the following contribution:

- Proposing a source code search engine for architectural tactics that relies on a novel text-based classification technique to automate the discovery, extraction and indexing of architectural tactics across 116,609 open source systems.
- Implementing a big data compatible architecture to search efficiently through 22 million source files.
- Utilizing information retrieval and structural analysis techniques to detect tactics and to identify the technical context in which the tactic has been used.

- Introducing a novel ranking algorithm to order the retrieved tactical files based on both tactic correctness and relevancy to the technical context stated in the users query.
- Conducting a series of experiments to prove that our techniques are fairly successful at finding tactics that are mentioned in the user query and outperformed existing source code search engines.

1.5 Organization of the Thesis

The organization of the thesis is as follows. The initial part of Chapter 2 discusses the background work. The later part of the chapter defines some related concepts. Chapter 3 proposes the solution to the problem. It describes the necessary definitions, techniques and algorithms of the proposed solution. The experiments are described in Chapter 4, and also evaluates the outcome of those experiments. Chapter 5 provides the conclusion and the possible future expansions.

Chapter 2

Background and Related Work

This chapter reviews the thesis primary area of focus, software architecture tactics, and give an overview of the previous work that has been done on code search.

2.1 Related Work

Over the years, several code search engines have been made to support different code search needs. Usually, these systems maintain a cached archive of source code and/or metadata about these artifacts and use a set of heuristics to define the relevance of an artifact which is subsequently used in their ranking algorithm. They differ on various aspects such as types of input supported (e.g. free text [2, 18], search queries inferred from source code [11, 41], etc), granularity level of produced output (such as functions [18], source files [2], code fragments [6, 12, 14], components [11, 41], etc), releasing approach (as a Web site [12, 17, 18], an Eclipse plugin [2, 3, 11, 14], etc), underlying code search technology and so on.

In this context, Prospector [14] and Sniff [6] are tools for obtaining code snippets in Java, i.e., fragments of a source file which perform a specific task. On one hand, Prospector is released as an Eclipse plugin, so it creates search queries on-the-fly from the code being developed in the Eclipse editor and outputs a set of recommended code snippets for developers to reuse [14]. On the other hand, Sniff gets free text as inputs and return code fragments based on merged data from API documentations and publicly available Java code [6]. Similarly to Prospector and Sniff, Kim et al [12] presents a code search engine which

outputs Java API documentations along with sample code snippets that use those APIs.

McMillan et al. proposed Portfolio for finding functions written in C/C++ and allowing users to navigate among these functions based on their call dependency [18, 20]. Its ranking approach adapts the PageRank algorithm to match functions based on the terms within the function itself and in the invoked functions and ranking them based on the frequency of usage (i.e. how many times a function is called). In spite of the fact that the portfolio has been proved to better meet the developer's needs for finding reusable code snippets, their focus is way too low-level (at the function level) and returns only C/C++ code.

Sourcerer [2, 3] is a code search engine for retrieving reusable open-source code. For fetching and ranking the results, it verifies structural properties, dependencies and entities within the source code files. Such structural analysis represents an improvement over a keyword-based matching of source files. Similarly to ArchEngine, it crawls public source code available in the Web, stores those files in a local code repository and extracts, for each file, a list of keywords storing them in an index. However, differently from ArchEngine, it is limited for only files written in Java and does not use any heuristics for extracting the best code snippets for any architectural tactic and technical problem.

For fulfilling the need of finding reusable higher-level artifacts, Code Conjurer [11] and Codebroker [41] are search engines for finding reusable components developed in Java. Both of them do not require that developers explicitly provide search queries to their engine, instead they perform the search based on parsed information from source code being written by developers. The difference is that Code Conjurer generates search queries from test cases whereas Codebroker analyses the comments in the code written by programmers. Another tool for higher-level searches is Exemplar [17] which focuses on finding executable Java software projects for reuse. Unlike ArchEngine, they only return Java components/projects and do not support any heuristic for optimizing results to address the demand for finding sample code for architectural tactics.

Invented by M. Merakhorli, Archie [26, 30], is one of the automated data mining tools

that is used to discover any architectural tactics used in the source code. It works by detecting any tactical elements in the architecture of the source code by using its internal set of codebase classifiers. These code classifiers have the ability to detect a variety of architectural tactics such as authentication, heartbeat, audit, role-based access control (RBAC), asynchronous invocation, scheduler, resource pooling, and secure session. All these code classifiers have been studied and prepared using open source code fragments that have been taken from hundreds of projects consisting of these architectural tactics.

Besides these code search engines proposed and developed by the research community, there are also proprietary engines for retrieving specific source code. Examples of such engines are Koders (now named as Open Hub Code Search), Krugle and so forth. In addition, some public repositories (e.g. GitHub and SourceForge) support the search of code snippets and/or software projects.

There are many differences when we compare ArchEngine with the search engines discussed previously. First, the techniques used in ArchEngine are mainly focused on finding source code related to architectural tactics while traditional code search engines usually do full-text search on the source code documents and their capability to perform architectural tactics searches is limited or inexistent. Second, they output code in a specific programming language (e.g Java) and focus on retrieving lower-level results (such as functions and code fragments). Even if the search engine outputs higher-level artifacts (i.e. components or projects), their search heuristics does not emphasize finding artifacts that satisfy quality requirements through implementing architectural tactics.

2.2 Software Architectural Tactics

A complex software systems architecture is cautiously built to fulfill various concerns and needs such as security, usability, safety, dependability, and other vital qualities. In order to fulfill and address such concerns, architects make use of architectural patterns, also known as styles, and architectural tactics to build the complete software system [23, 24]. The foundation of the entire software architecture is done using the architectural tactic, which

comes in various shapes and sizes to tackle a range of quality issues . The usability of architectural tactics can be seen especially in fault tolerant and high performance systems. On one hand where reliability tactics such as redundancy with heartbeat, voting, and check pointing provides some solutions for mitigating, detecting, and recovery of faults, on the other, performance tactics such as resource pooling and scheduling helps in optimizing response time and latency[28].

The focus of this research is limited to 14 specific tactics, in order to keep the scope of this work manageable. They were selected based on representation of an array of reliability, performance, and security requirements. These specific tactics defined as follows [7]:

- **Active Redundancy:** Configuration where in all of the nodes(active or redundant spare) in a protection group receive identical inputs in parallel, so recovery and repair can occur in milliseconds.

- **Audit Trial:** A copy of each transaction and associated identifying information is maintained. This audit information can be used to recreate the actions of an attacker, and to support functions such as system recovery and non-repudiation.

- **Authentication:** Ensures that a user or a remote system is who it claims to be. Authentication is often achieved through passwords, digital certificates, or biometric scans.

- **Check Point:** Recording of a consistent state created periodically or in response to a specific event.

- **HeartBeat:** A reliability tactic for fault detection, in which one component (sender) emits a periodic heartbeat message while another component listens for the message (receiver). The original component is assumed to have failed when the sender stops sending heartbeat messages. In this situation, a fault correction component is notified.

- **Kerbrose:** A network authentication protocol. It is designed to provide strong authentication for client/server applications by using secret-key cryptography.
- **Load Balancing:** An application of the scheduling resources tactic would ensure that one broker is not overloaded while another one site is idle.
- **PBAC:** Use of digital policies compromised of logical rules, to guide authorization decision.
- **Ping Echo:** An asynchronous request/response message pair exchanged between nodes that is used to determine the reachability and round-trip delay through the associated network path.
- **Resource Pooling:** Limited resources are shared between clients that do not need exclusive and continual access to a resource. Pooling is typically used for sharing threads, database connections, sockets, and other such resources. This tactic is used to achieve performance goals.
- **RBAC:** User/Process Authorization is used to ensure that an authenticated user or remote computer/process has the rights to access and modify either data or services.
- **Scheduler:** Resource contentions are managed through scheduling policies such as FIFO (First in first out), fixed-priority, and dynamic priority scheduling.
- **Secure Session:** Allows an application to only require the users to authenticate once to confirm that the user requesting a given action is the user who provided the original credentials.
- **Validation Interceptor:** Checks if there is any validation errors or not, used to get information about the error messages defined in the action class.

2.3 Inverted Index

Inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents [1]. The inverted index is the most widely used index model at present. All words in the documents are indexed as keywords in inverted index. The recording item for each word includes the documents that contain the word, as well as its location in the document. Thus, when user search a word in the index, he can easily find the document which contains the word and its location in the document. For inverted index of search engine, since the number of web pages related to lexical items is dynamically changed, and so is the content of the web pages, it is more difficult to maintain the inverted index. However, inverted index has great advantage in query system. Inverted index is widely used in the system, which has high demand for the response time of searching, since one may find all document information that contains the word in just one search. Many researchers have discussed the key technologies of inverted index [13]. The inverted index is an indexing mechanism for words. It can improve search speed. Each index entry in the inverted index is composed of index term and its appearance, each index term has a posting list to record all information of the word appearing in the documents. This information contains the index ID, the position in the document and the index appearing frequencies, etc [36].

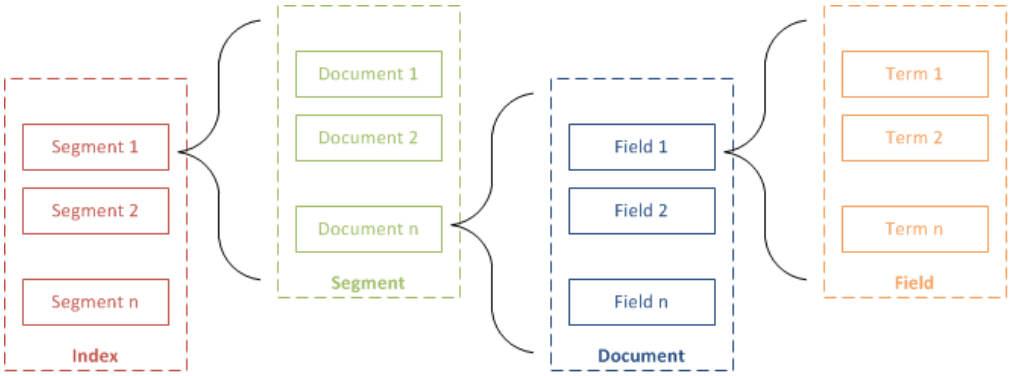


Figure 2.1: Inverted Index Structure

In this work, we use one of the most high-performance, scalable, full-featured, open-source inverted index libraries named Apache Lucene [8]. Apache Lucene is a search library written in Java. Its popular in both academic and commercial settings due to its performance, configurability, and generous licensing terms. Lucene index, as shown in Figure 2.1 is a set of documents that are to be searched. The index may be composed of multiple sub-indexes, or segments. Each segment is a fully independent index, which could be searched separately. A document is essentially a collection of fields. A field consists of a field name that is a string, and one or more field values. Fields are constrained to store only one kind of data, binary, numeric, or text data. There are two ways to store text data: string fields store the entire item as one string; text fields store the data as a series of tokens,the text is broken up into terms at index time. Lucene provides many ways to break a piece of text into tokens, as well as hooks that allow developers to write custom tokenizers. We built our search engine on top of Lucene index that contains all source files in our repository. In section 3.3 we discuss in detail how this index was built.

2.4 Vector Space Model

Vector space model is an algebraic model used by search engines to rank matching documents according to their relevance to a given search query. VSM is a bag-of-words retrieval technique that ranks a set of documents based on the terms appearing in each document, as well as the query. In the statistically based vector-space model, a document is conceptually represented by a vector of keywords extracted from the document, with associated weights representing the importance of the keywords in the document and within the whole document collection; likewise, a query is modeled as a list of keywords with associated weights representing the importance of the keywords in the query.

The weight of a term in a document vector can be determined in many ways. A common approach uses the so called tf-idf method, in which the weight of a term is determined by two factors: how often the term j occurs in the document i (the term frequency $tf_{i,j}$) and how often it occurs in the whole document collection (the document frequency df_j).

Precisely, the weight of a term j in document i is

$$w_{i,j} = tf_{i,j} * idf_j = tf_{i,j} * log_t(N/df_j) \tag{2.1}$$

where N is the number of documents in the document collection and idf stands for the inverse document frequency. This method assigns high weights to terms that appear frequently in a small number of documents in the document set. Once the term weights are determined, we need a ranking function to measure similarity between the query and document vectors. A common similarity measure, known as the cosine measure, determines the angle between the document vectors and the query vector when they are represented in a V-dimensional Euclidean space, where V is the vocabulary size. Precisely, the similarity between a document D_i and a query Q is defined as

$$sim(Q, D_i) = \frac{(\sum_{j=1}^{v} w_{Q,j} * w_{i,j})}{\left(\sqrt{\sum_{j=1}^{v} w_{Q,j}} \cdot \sqrt{\sum_{j=1}^{v} w_{i,j}}\right)} \tag{2.2}$$

where $w_{Q,j}$ is the weight of term j in the query, and is defined in a similar way as $w_{i,j}$ (that is, $tf_{Q,j}\ \ idf_j$). The denominator in this equation, called the normalization factor, discards the effect of document lengths on document scores. Thus, a document containing $\{x, y, z\}$ will have exactly the same score as another document containing $\{x, x, y, y, z, z\}$ because these two document vectors have the same unit vector. We can debate whether this is reasonable or not, but when document lengths vary greatly, it is crucial to to take them into account.

Chapter 3

Design

3.1 Overview of Approach

The architecture of our search engine and its components are depicted in Figure 3.1. The first component is an *ultra-large-scale source code repository*, which contains over 116,609 open-source projects extracted from various online software repositories. The second component is our novel *source code indexing* technique, which represents projects and their source files in a form of index that is efficient for performing information retrieval techniques. The third component is a *tactic detector* [21, 31] capable to detect various architectural tactics in the indexed code artifacts. The tactic detector relies on information retrieval techniques, and its accuracy was previously validated in a series of experiments [21, 31].

The fourth component is a *dependency analyzer*, which generates a dependency matrix for each tactical file in the source code of a project. This matrix is then used by the fifth component - *Matching Technical Problem* - to find whether the implementation of a given tactic is related to a technical problem/context or not. Technical context refers to a framework, technology, programming language, or APIs which can be used to implement the tactic or technical problem in which the tactic needs to be implemented.

The final component is a novel *Ranking algorithm*, which ranks the source files in the search results based on; (i) the semantic similarity of a source file to a searched tactic (ii) the semantic similarity of a source file and its direct dependent files to a technical problem represented in the search query.

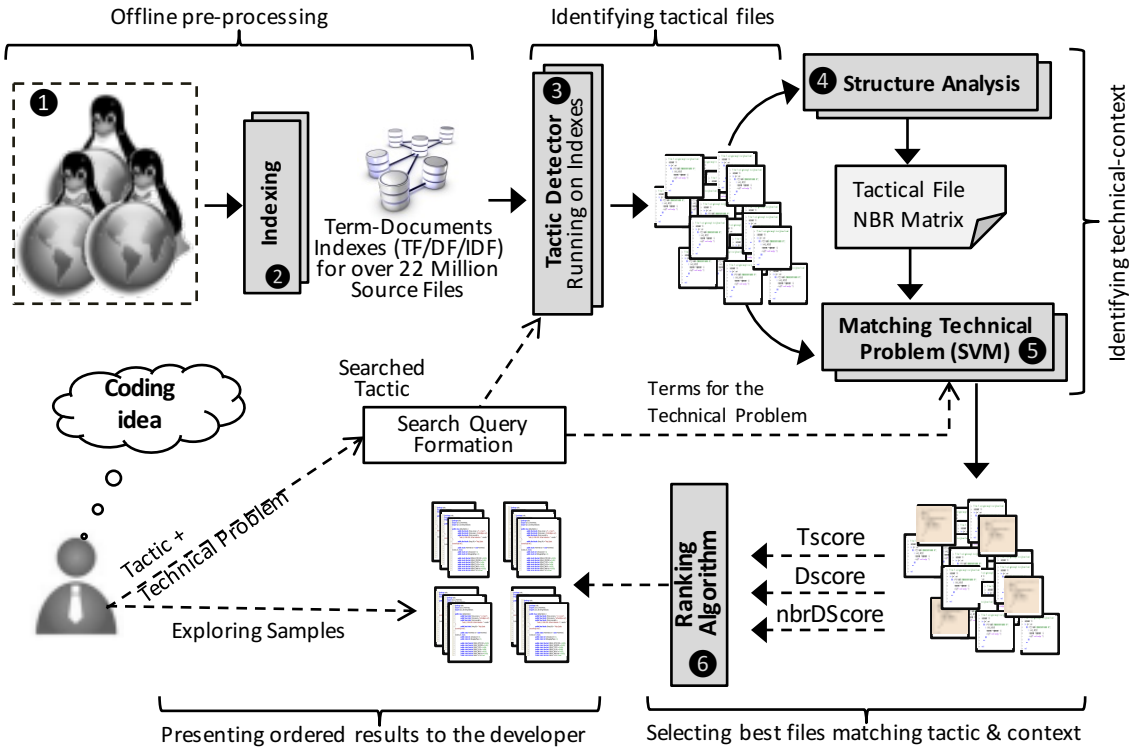


Figure 3.1: The architecture of our search engine

The search process is initiated when a user provides a preliminary description of the tactic implementation problem represented in the form of a query. Examples of such inquiries are provided in Figure 1.1. This description is used to initiate the search query composing of *desired tactic* and *technical context* in which the tactic should be implemented. The user is asked to separate the tactic and technical problem. For instance, when a developer is searching for "sample implementations of Heartbeat tactic when it is used in a multi-threaded program to monitor HTTP type processes", the query will have the following two pieces, (Tactic, Heartbeat) and (Technical_Problem, Multi-threaded HTTP process). ArchEngine breaks this query into two pieces of tactic-and-problem so it can retrieve the cluster of files implementing the tactic first. Then, it filters these files based on how similar each cluster of tactical files is to the requested technical problem. A tactic-similarity score and context-similarity score will be calculated for each file (described in section 3.6). Subsequently, the ranking algorithm order the results based on these two metrics.

3.2 Creating Ultra-Large Scale Repository of Open Source Projects

The first component is a large scale repository of software projects extracted from online open-source repositories. The current version of our repository contains 116,609 projects extracted from GitHub, Google Code, SourceForge, Apache, and other software repositories. We have developed different *code crawling* applications to retrieve projects from all these code repositories. To extract the projects from GitHub, we make use of a torrent system known as GHTorrent[1] that acts as a service to extract data and events and gives it back to the community in the form of MongoDB data dumps. The dumps are composed of metadata about projects such as users, comments on commits, programming languages, pull requests, follower-following relations, and others.

[1] http://ghtorrent.org/

We also used *Sourcerer* [38], an automated crawling, parsing, and fingerprinting application developed by researchers at the University of California, Irvine. *Sourcerer* has been used to extract projects from publicly available open source repositories such as Apache, Java.net, Google Code, and Sourceforge. Its repository contains versioned source code across multiple releases, documentations (if available), project metadata, and a coarse-grained structural analysis of each project. We downloaded the entire repository of open source systems from these code repositories.

Table 3.1: Overview of the projects in ArchEngine's Source Code Repository

Language	Freq.	Language	Freq.	Language	Freq.
Java	32191	Matlab	354	Scheme	80
JavaScript	22321	Arduino	321	Prolog	77
Python	9960	Emacs Lisp	321	F#	74
Ruby	8723	Rust	308	D	72
PHP	8425	Puppet	286	Pascal	60
C++	5271	Groovy	253	FORTRAN	45
C	4592	SuperCollider	185	Racket	44
C#	4230	Erlang	154	VHDL	43
Objective-C	2616	Visual Basic	134	Verilog	43
Go	1614	ActionScript	120	Bison	39
CoffeeScript	1187	OCaml	105	Cuda	37
Scala	729	Assembly	98	Objective-C++	33
Perl	699	ASP	85	SQF	26
Lua	458	Dart	84	Mathematica	25
Clojure	456	Julia	84	Apex	22
Haskell	456	Elixir	82	PureScript	22

*Total number of projects:116,609, *Total number of source files: 23M

Having extracted all these projects from GitHub and other repositories, we performed a data cleaning in which we removed all the empty or very small projects (i.e. projects that have less than 20 source files). Table 3.1 shows the frequency of all the projects in different programming languages in our repository as well as its size in terms of number of projects and source code files.

3.3 Source Code Indexing

ArchEngine uses text-mining techniques to identify and retrieve tactical code snippets. This requires efficient indexing of terms across all the source files in our ultra large scale repository. The second component of ArchEngine is a term-document indexing module, which

indexes the occurrence of terms across source files of each project in our code repository. This component, known as *Indexing*, first pre-processes each source file in which it uses: (i) a stemmer to reduce words to their root forms, the stemming task was performed using Porter's Stemming Algorithm [40], (ii) a stopper to remove common terms, (iii) a splitter that splits variable names based on the common coding conventions. After these preprocessing, the source files are indexed.

The index stores statistics about each documents (source files) such as *term frequency (TF)*, *document frequency (DF)*, *TF/IDF* and *location of source file* in order to make term-based search more efficient. This is an inverted index, which can list the source files that containing a specific term [16]. Furthermore, the index stores the metadata (language, project etc.) for each source file.

The indexing process is the core function of ArchEngine that is used during identification of tactic and indexing tactical files, searching, and other associated tasks such as highlighting, querying, language analysis, and so forth. All the files that were retrieved from the earlier step are given as inputs to the indexing system. The ArchEngine index is based on the popular Apache Lucene [16] information retrieval engine. The index model equates a Lucene document to every source file in the repository. A document is made up of a collection of fields, each field being a name/value pair. The simplest form of value is a collection of terms, where a term is the basic unit for search and retrieval. Terms are extracted from various parts of a source file, and stored in the fields of the document corresponding to that file. ArchEngine index model can be categorized into five fields:

- **Id**: Stores the full path of the source file in the repository.
- **Dependencies**: Stores the dependency matrix of that file.
- **Contents1**: Stores the actual contents of the source file in the repository. This field was used to match the tactical files.
- **Contents2**: Stores the actual contents of source files and all theirs neighbouring files. This field was used in matching the technical context.

- **Language**: This field stores the programing language used in that file.

3.4 Detecting Architectural Tactics

To identify which source codes from our repository are architecturally-relevant (i.e. implements an architectural tactic), we use the tactic detection algorithm developed previously [32]. This technique uses a custom-made supervised machine learning algorithm trained with manually collected code snippets that implement an architectural tactic. This detection algorithm encompasses three phases: *Data Preparation Phase*, *Training Phase* and *Classification Phase*. These phases work as follows:

- *Data Preparation Phase*: In this phase, the training set is preprocessed using standard information retrieval methods. In this preprocessing, the stop words (i.e. irrelevant words, such as programing language keywords) are removed and the identifiers are split into their primitive parts. Subsequently, those splitted identifiers are stemmed in order to find their root forms. Lastly, the source codes are broken down into a list of terms which are used in the next phase.

- *Training Phase*: As the name suggests, in this phase the classifier mechanism is trained with the list of terms extracted in the previous phase from the manually established dataset of code snippets that implement a tactic. From this training data, the training mechanism obtains a list of *indicator terms*, i. e., terms that are a representative for the tactic. Also, a *weight value* is given to each *indicator term*. This *weight value* shows the level of importance of an indicator term with respect to the tactic. For example, the term ”role” is one of the most used terms when implementing the ”Role-Based Access Control”, so it receives a higher *weight value*.

  A formal definition is given as follows: let q be a tactic of interest (e.g. *Heartbeat*). The *indicator terms* of the tactic q are mined by considering the set S_q of all classes within the training set that are related to the tactic q. The cardinality of S_q is defined as N_q. Each term t is assigned a weight score $Pr_q(t)$ that corresponds to the

probability that a particular term t identifies a class associated with tactic q. The frequency $freq(c_q, t)$ of the term t in the class description c related with the tactic q, is computed for each tactic description in S_q. Then, the $Pr_q(t)$ is computed as:

$$Pr_q(t) = \frac{1}{N_q} \sum_{c_q \in S_q} \frac{freq(c_q, t)}{|c_q|} * \frac{N_q(t)}{N(t)} * \frac{NP_q(t)}{NP_q} \tag{3.1}$$

- *Classification Phase*: In this phase, the *indicator terms* of an architectural tactic (calculated in the *Training Phase* using the Equation 3.1) are used to calculate the *probability score* ($Pr_q(c)$) which indicates the likelihood that a given source code c is associated with the tactic q. Let I_q the set of *indicator terms* for the tactic q identified during the training phase. The classification score that class c is associated with tactic q is then defined as follows:

$$Pr_q(c) = \frac{\sum_{t \in c \cap I_q} Pr_q(t)}{\sum_{t \in I_q} Pr_q(t)} \tag{3.2}$$

  where the numerator is computed as the sum of the term weights of all type q indicator terms that are contained in c, and the denominator is the sum of the term weights for all type q indicator terms. The probabilistic classifier for a given type q will assign a higher score $Pr_q(c)$ to a source code c that contains several strong indicator terms for q. Source codes are considered to be related to a given tactic q if the classification score is higher than a selected threshold.

  The threshold value is established through the 10-fold cross-validation process [5], a standard approach commonly used in software engineering research to evaluate accuracy and generalizability of data mining techniques. In this process, there are ten groups in which each of them contains one architectural-related code snippet and four unrelated ones. The system repeatedly is trained with nine of these groups and the remaining one is the testing set (i.e. it has the five source codes classified into architectural-related or unrelated). This execution is repeated until all groups are used as testing sets for a variety of threshold values. Since we know from the testing set which files are related/unrelated to an architectural tactic, we can verify which

threshold value has a better performance in the detection accuracy of architectural tactics. The accuracy of tactic detection has been previously evaluated in a number of extensive studies [31, 21]. Currently our approach is able to accurately detect over 14 architectural tactics such as *heartbeat*, *scheduling*, *resource pooling*, *authentication*, *authorization*, *secure session management*, *ping-echo*, *checkpointing* and *audit trail*, *Role-based access control (RBAC)* [31, 29, 27, 21].

3.5 Matching Technical Problem

Until previous step, ArchEngine was able to detect tactical files across our ultra large scale repository, In the next two steps, it will calculate a score for the technical-context in which the tactic is implemented. This would help us not only identify the tactical file but also separate tactic instances which are implemented using technologies or deal with the technical problems stated in the developers' query.

The technical context can be discovered from the areas of the code where a tactic is adopted. To discover the technical-context, ArchEngine uses the latent-topics within the tactical file itself and neighboring files which use or provide utilities for the tactical file. This is done because the technical context is not fully presented in the tactical file itself and is reflected in the surrounding files. For example, in case of the Authentication tactic, the files which use the authentication function describe the technical context rather than the files which implement the authentication. There might be cases the technical-context can be observed in both tactical file and the neighboring files which have direct method call with the tactical file. Therefore, ArchEngine needs to identify the technical context for the tactic by looking at the files which interact with the tactical file.

Therefore, the *Structural Analysis* component is used to find the source files that have direct method call with the tactical files. This component is responsible for statically analyzing all projects in ArchEngine's repository and generating a call graph represented in the form of dependency matrix for each tactical file.

The dependency matrices extracted for each tactics are used by the next component

- *Matching Technical Problem* - which implements a paralleled version of Vector Space Model (VSM) [35] to calculate a score for the relevance of the tactic's technical-context and what is stated in the developer's query. The developer's query is broken into two parts: the tactic under search and the technical context. The second part of the query is used by this component to calculate a score for the tactic's technical context.

This component is capable of running over 22 million source files in a few seconds. Vector Space Model (VSM), is a standard approach typically used by search engines to rank matching documents based on their relevance to a given search query.

In the VSM, the developer's query (technical problem part) q and each source file f is represented as a vector of terms T = $t_1, t_2, ..., t_n$ defined as the set of all terms in the set of queries. Therefore, a source file f is represented as a vector $\vec{f} = (w_{1,f}, w_{2,f}, ..., w_{n,f})$, where $w_{i,f}$ represents the weight of the term i for source file f. A query is similarly represented as $\vec{q} = (w_{1,q}, w_{2,q}, ..., w_{n,q})$. The standard weighting scheme known as $tf - idf$ is used to assign weights to individual terms [35], where tf represents the term frequency, and idf the inverse document frequency. Term frequency is computed for source file f as $tf(t_i, f) = (freq(t_i, f))/(|f|)$, where $freq(t_i, f)$ is the frequency of the term in the document, and $|f|$ is the length of the document. Inverse document frequency idf, is typically computed as:

$$idf_{ti} = log_2(n/n_i) \tag{3.3}$$

where n is the total number of source files in the tactic collection, and n_i is the number of source files in which term t_i occurs. The individual term weight for term i in source file f is then computed as $w_{id} = tf(t_i, f) \times idf_{ti}$. A similarity score $ContextSim(f, q)$ between source file f and technical query q is computed as the cosine of the angle between the two vectors as

$$ContextSim(f, q) = \frac{(\sum_{i=1}^{n} w_{i,f} w_{i,q})}{\left(\sqrt{\sum_{i=1}^{n} w_{i,f}} \cdot \sqrt{\sum_{i=1}^{n} w_{i,q}}\right)} \tag{3.4}$$

The similarity score between the technical part of the query and the topics in tactical file and its neighboring files is used as a score for relevance of the technical context. This

score is used to identify the source files that are relevant to the technology used alongside with the tactic. In the next section, we present a formula for ranking the results based on the tactic and technical scores.

3.6 Ranking Algorithm

To rank the results of our search engine, a custom ranking algorithm has been developed. There are three components that compute different scores in the ArchEngine's ranking mechanism presented in formula 3.5: (i) a component that computes a tactical score for a given file, reflecting the probability that a source file implements a tactic ($TScore(f, t)$, calculated using formula 3.2, where f is a source file and t is a tactic,), (ii) a component that computes a score called $ContextSim(f, q)$ for the similarity of q, a technical problem queried by the user and content of the tactical file, f. This score is calculated based on word occurrences and cosine similarity formula described in equation 3.4. Lastly, (iii) a component that computes an average technical similarity score for all the files interacting the tactical file (nbr: all the neighboring files for f). This last component provides a score for the context in which the tactic has been adopted. This is particularly important since in some tactics, the frameworks or technology used by the developers are not implemented in the same files. Developers sometimes separate the tactical functions and the contextual concepts where the tactic is implemented with.

The total ranking score is the weighted sum of these components. Each component produces results from different perspectives (i.e., tactical matches, direct technological matches, indirect technological matches). Our goal is to produce a unified ranking by putting these orthogonal, yet complementary, rankings together in a single score. To do so, we compute the rank of a result for a given tactic and search query as follows:

$$rank(f, t, q) = TScore(f, t) + ContextSim(f, q) + \frac{\sum_{d \in nbr(f)} ContextSim(d, q)}{nbr(f)} \tag{3.5}$$

3.7 Search Process

A developer initiates the search process by first selecting the desired tactic. He then proceeds to specify the problem where the tactic is used to address the technology, or framework used to implement the tactic. Separating these two pieces will help ArchEngine to better identify the context in which the tactic is implemented and return the results which match the described in the query. Examples of such queries are:

Query#1: `Heartbeat implementation over UDP socket programming`
Query#2: `Secure session management using HttpSession of Java`
Query#3: `Thread pooling multi-thread implementation executor service of java`

Chapter 4

EXPERIMENT AND EVALUATION

We conducted some experiments with the system we developed. The goal for this experiment was to evaluate how our proposed search technique performed. In the following sections, we discuss more about our experiments.

4.1 Evaluation and Comparison with State-of-the-Art

A set of experiments were conducted to compare the performance of ArchEngine against other generic code search engines. For this purpose, we evaluated ArchEngine against Open Hub [1] (which used to be known as Koders), Krugle [2], SourceForge [3] and the built-in search in GitHub repository [4]. These Web systems were chosen as representative samples of code search engines used periodically by developers when performing their coding activities.

We believe ArchEngine, like many other code search engines, is useful for junior developers and those with less programming experience. Senior developers, who are familiar with architectural tactics and technologies, are less likely to search for sample code snippets to get implementation ideas. Therefore, to evaluate the practicality of ArchEngine, we recruited graduate students as subjects who are familiar with architectural concepts but do not necessary have extensive experience as architect or software developer.

[1] `https://code.openhub.net/`
[2] `http://opensearch.krugle.org/`
[3] `http://sourceforge.net/`
[4] `https://github.com/search/`

Any code search engine needs to be evaluated regarding the accuracy of the items in the results, as well as their ranking. In the next subsections, we detail each step performed to execute this experiment, the metrics that were collected for assessing the performance of the tools, and the results we obtained.

4.2 Methodology

This experiment involved a total of 21 subjects enrolled in a graduate Software Architecture course. They were asked to complete three *architecture prototyping* tasks. The subjects were required to implement a minimal functional system and satisfy a quality requirement (availability, security or performance) through the usage of specific architectural tactics. The tactics under consideration were: *Heartbeat*, *Secure Session Management*, and *Thread Pooling*. A detailed discussion about these tactics is presented in [4]. Although the current version of our search engine supports 14 architectural tactics [21], for the sake of evaluating the ArchiEngine's proof of concepts, we randomly selected the above 3 tactics out of 14. Comparing multiple search engines for multiple tactics will be very time consuming, we believe similar results will be obtained for the other tactics.

The subjects were asked to use ArchEngine and other provided code search engines to find reusable sample source files or get ideas to implement the three tactics. The search process was performed in pairs. Each pair of subjects created a set of queries containing a list of keywords to search for architecturally relevant code snippets that could help them complete the assigned tasks. These queries were applied to all the search engines. Later, each code snippet returned was analyzed in terms of its *correctness*. A search result was considered correct if it contained an implementation of the architectural tactic within the queried technical context or frameworks of interest.

Table 4.1: Tasks assigned to students in the experiment

Task#1: Implement Heartbeat	**Task#2:** Implement Pooling	**Task#3:** Implement Secure Session Management
Task Description: Implement Heartbeat Tactic. The implementation is minimum prototyping of the tactic than full implementation of a system. **Considerations:** • Select a domain • Develop a critical process (with minimum functionality) • Design a Non-deterministic failure in this process which makes it crash. • Implement Heartbeat to monitor the process • Heartbeat implementation should have all the required fault detection features. • Implement send/receive/monitoring functions on different processes **Constraints:** Languages are not limited. You can use the existing frameworks that offer fault detect APIs or implement the tactic from scratch.	**Task Description:** implement "thread Pooling" Tactic. The implementation is minimum prototyping of the tactic than full implementation of a system. **Considerations:** • Select a domain • Develop a performance critical task (with minimum functionality) • Create a pool of threads to accomplish that performance critical task • Pool size should be 10 threads. • Please develop any hypothetical tasks. Examples could be: "processing files", "Crawling web-content" or "generating prime numbers". • Make sure your task would take more than 10 threads to accomplish it. **Constraints:** Languages are not limited. You can use the existing frameworks or implement from scratch.	**Task Description:** implement "Secure Session Management" Tactic. The implementation is minimum prototyping of the tactic than full implementation of a system. **Considerations:** • Select a domain • Create a light web application. • implement secure session management with mock-tasks. • The web application check if the user is already authenticated and is authorized to do requested operation, and then proceed with the operation on behalf on the user. • Have at least three users, and one task. One user is authorized to perform the task, one is not, and the last one is not authenticated. **Constraints:** Languages are not limited. You can use the session management APIs of any existing library.

4.3 Assigned Architecture-Prototyping Tasks

Table 4.1 shows the assigned tasks to the subjects. Each task includes an architecture-prototyping to implement basic components of a software system, mock business components with minimum functions but fully implement the tactic. For each of these tasks, we provided generic considerations and constraints that partially define how the tactic should be developed (yet, subjects had the flexibility to select their software type and technologies).

The following three tasks were performed by subject teams (8 pairs and 4 individuals).

Task#1: Fault detection using Heartbeat. In the first task, subjects were asked to develop a dependable system that could detect the failures of a critical component using the *Heartbeat* tactic. Subjects were required to implement all the classical elements of heartbeat: (i) a (*heartbeat sender*) process that emits a periodic heartbeat message to indicate its availability, (ii) a (*heartbeat receiver*) which checks the availability of the sender and lastly (ii) a (*heartbeat monitoring*) process which imitates the recovery in case of detecting a failure [25].

Task#2: Resource arbitration & performance enhancement using Pooling. In the second task, the subjects were asked to develop the *Thread Pooling* tactic to improve the performance of a system. The requirement is that the system shall have resources that are expensive to create, execute and maintain. Thus, the functionality is broken down into chunks of executable units which are added to the pool of threads.

Task#3: Web-based secure session management. The third task requires the development of a Web application with at least two tasks in that can only be completed by different groups of authenticated and authorized users. To keep track of authentication and authorization data, subjects needed to implement the secure session management.

Due to the extensive cost of implementing a system, subjects were asked to (i) fully implement the tactic, (ii) develop the functional features as needed, (iii) implement mock-components for the remaining domain components and features.

These subjects were advised to perform the assigned tasks either in pairs or individually.

Furthermore, they could develop a tactic from scratch or choose an existing framework to implement it. They had the flexibility of working in any application domain. We presumed that such freedom of choice would allow our search engine to be tested against a variety of scenarios, thereby avoiding biases due to underlying technologies and software domains. The subjects used the search engines introduced in previous sub-section to find sample code snippets for their architecture-prototyping tasks.

4.4 Hypothesis

In this experiment, we aim to evaluate the following hypothesis:

> ***Hypothesis**: ArchEngine users find more relevant architectural code snippets compared to the users of Open Hub, Krugle, SourceForge and the GitHub search.*

Manually evaluating all the search results is not feasible because it requires a lot of time to be completed. In the context of Web search, individuals are unlikely to go deeper in the results of a search [15]. Therefore, we asked the subjects to analyze only the ten topmost results with respect to their *correctness*, i. e., if the returned code snippet implements the tactic using the technology under consideration.

The metrics described in the next section are used to verify whether there is a statistically significant difference between ArchEngine and existing search engines.

4.5 Evaluation Metrics

In the context of traditional information retrieval, *precision* and *recall* are commonly used for evaluating the performance of a system. However, for web-scale information retrieval techniques such as source code search engines, recall is no longer a meaningful metric, as many queries have thousands of relevant source files, and few developers will be interested in reading all of them. Instead, *Precision at k* (P@k) is recognized as a useful metric and

widely used by researchers (e.g., P@10 or "Precision at 10" corresponds to the number of relevant results on the first search results page). In our first experiment we report P@10.

However, reporting precision at K is not enough for evaluating search engines. This metric fails to take into account the ranking of the results, i.e., whether the relevant source files are placed in the topmost positions. Therefore, besides using the *P@10* to evaluate the performance of the search engines, we also calculated the Normalized Discounted Cumulative Gain (NDCG)- a metric which considers not only the relevance of a returned code snippet but also its order in the result set. These two metrics, *P@10* and *NDCG*, are computed as follows:

$$P@k = \frac{|\{relevant\ tactical\ files\} \cap \{top\ K\ items\ in\ the\ results\}|}{k} \tag{4.1}$$

Given that we analyze only the first ten results, the value of k is equals to 10 in the equation. This formula shows the accuracy of the search engines, the next formula is used to examine the power of search engines in ordering the results.

$$NDCG = \frac{rel_1 + \sum_{i=2}^{n} \frac{rel_i}{log_2(i)}}{NF} \tag{4.2}$$

where rel_i in this equation is a binary function that indicates the correctness of the result (it is equals to 1 only when the result is *correct*, otherwise it yields zero). Given that we analyze only the first ten results, the value of n is equals to 10. The NF in Eq. 4.2 is a *normalization factor* equals to the highest possible value achieved when all the results are correct (i.e. NF = 1 + 1/$log_2(2)$ + 1/$log_2(3)$... + 1/$log_2(10)$ $\approx$ 5.25449).

These two metrics are commonly used together to evaluate the results of web-based search engines.

4.6 Results

Figure 4.1 shows our experiment findings after evaluating the top 10 answers returned by our system and other search engines. As previously mentioned, we compared ArchEngine

with Koders, Krugle, and GitHub in terms of their *P@10* and the *NDCG*. We use a Box plot to verify how metrics are distributed for each tactic. In particular, it shows the highest and lowest values, as well as the median and the quartiles. As shown in this figure, in all the three tasks assigned to the subjects in the Software Architecture class, the results of ArchEngine outperformed other search engines.

Regarding the P@10, we calculated the mean *P@10* achieved for all 12 distinct queries evaluated by the subjects for a given tactic. We can see that ArchEngine achieved a value of 0.63, 0.8 and 0.72 for Heartbeat, Secure Session Management and Thread Pooling, respectively. These three values outperformed the other search engines, being at least 15% higher than the other code search engines. Specifically, Thread Pooling had an outstanding performance with its P@10 43%, 49% and 59% higher than Koders, Krugle and GitHub, respectively.

ArchEngine was also able to achieve a better search ranking, having the correct links mostly appearing on top of the list. Figures 4.1(a), 4.1(c) and 4.1(e) depict the box plots for all the three tactics. NDCG value for ArchEngine was less dispersed, which means that ArchEngine had a consistently good performance for the evaluated tactics. Moreover, ArchEngine had higher NDCG values in most of its query results. In the case of Heartbeat, even though the median is almost the same for ArchEngine and Koders, we can see that the Koders' results are more dispersed. Koders' box plot is taller than ArchEngine's ranging from 0.15 to 0.69, whereas most of ArchEngine's results fall into the range of 0.4 to 0.8. The results of this experiment provide a positive answer to our research hypothesis, and clearly show the feasibility and accuracy of our search engine in extracting and returning tactic related code for a given technical problem.

4.7 Threats to Validity

In this section. we discuss the threats to the validity of our work. as well as the ways we attempted to mitigate them. These threats are classified into *internal* and *external* validity. On one hand, the *internal validity* refers to what extent a research study reduced systematic

errors and biases in order to draw conclusions about cause-and-effect relationships from the data collected. On the other hand, the *external validity* is concerned with the level to which our work is generalizable.

4.7.1 Internal Validity

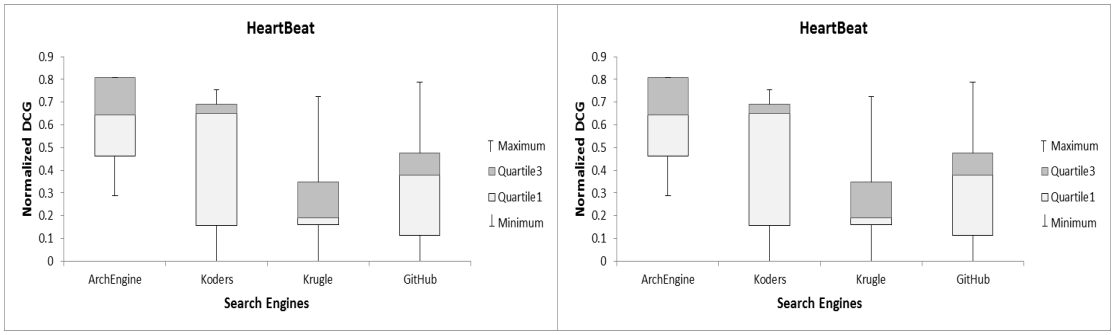
In respect to the design of ArchEngine, the greater threat lies in the fact that we used manually collected code snippets to train our tactic-classifier (the component responsible for identifying which source codes are architecturally-relevant). Thus, the identification of architectural code snippets was dependent upon the knowledge of the individuals who collected the training data. Therefore, it is possible that our search engine does not find additional implementation of tactics which used an entirely different terminology from the training data. Despite the challenge of solving this threat, we partially mitigated this risk by carefully locating real examples of implementation of tactics through systematically searching, browsing and analyzing, with the aid of expert's opinion, the code snippets we found in public source code repositories.

Regarding our experiment design to evaluate ArchEngine, the main threat came from the participants' perception on judging the correctness of a search result. To evaluate the correctness attribute of a search result, the individual should have a solid knowledge on the purpose of the tactic and have an overall understanding on how these tactics are implemented in the source code. Thus, if an individual does not have a full understanding on architectural tactics, as well as how to develop it using a given programming language and/or framework, then it can lead to biased results. We mitigate this threat by using students that were enrolled in a graduate Software Architecture course which were previously taught about these architectural tactics with class diagrams and real examples on how to apply those tactics in real systems. Moreover, since they were allowed to use any programming language and underlying frameworks/APIs, this gave freedom to the students to choose the technologies they were most comfortable with, so reducing the risk of wrong analysis due to lack of technical background.

4.7.2 External Validity

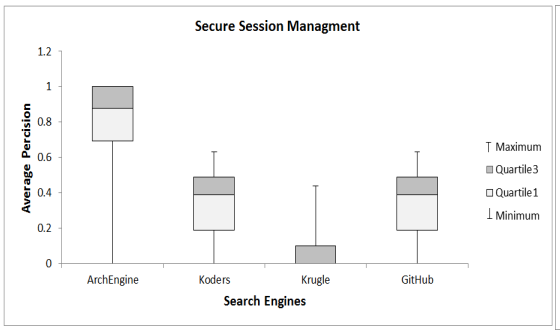
The main threat to the external validity of this work is that our search engine was evaluated for three architectural tactics mostly implemented with the following technologies: C#/ASP.net, Java, PHP and Python. However, since the results demonstrated a good performance for these subsets of technologies that were randomly chosen by students, it gives us confidence that the results would not be significantly different if the same experiments were performed for other programming languages and frameworks.

Another threat to our work is that, despite we have already downloaded over 100,000 of projects, yet our local repository is smaller than existing ones. For example, SourceForge claims to host about 430,000 projects. However, our repository is continuously being increased through downloading and indexing the source codes from public source code repositories available on the Web. Thus, despite the relative small size of our repository, it is increasing over time and yet, we were still able to get satisfactory results.

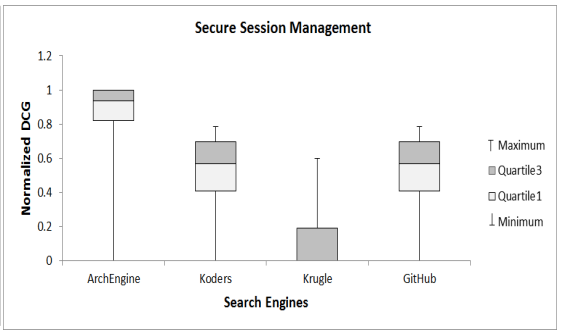


(a) Average Precision results for Heartbeat

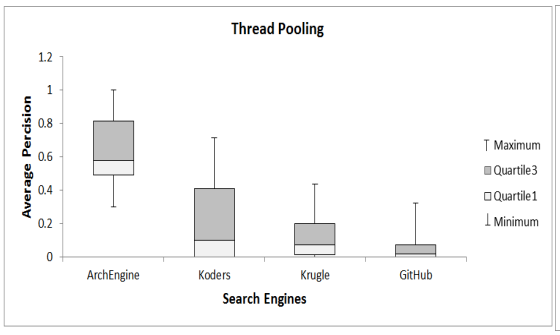
(b) NDCG results for Heartbeat



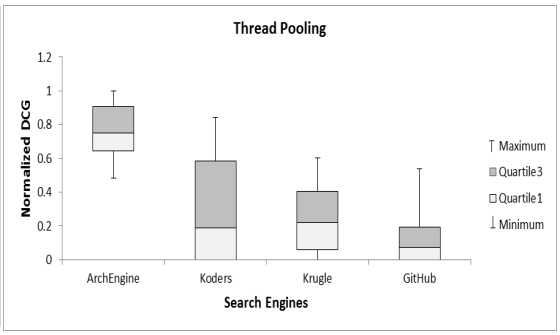
(c) Average Precision results for Secure Session Management



(d) NDCG results for Secure Session Management



(e) Average Precision results for Thread Pooling



(f) NDCG results for Thread Pooling

Figure 4.1: Comparison of ArchEngine's performance with state-of-the-art code search engines

Chapter 5

Conclusions

5.1 Conclusion

We created an approach called ArchEngine for finding highly relevant source files that implement an architectural tactic within a given technical context. ArchEngine uses an extensively large code base repository. In ArchEngine, we combined various data mining, information retrieval, and indexing techniques with a light weight source code analysis approach to retrieve tactical files adopted within a specific technical context or developed using a particular technology. Furthermore, we developed a novel ranking algorithm to sort the search results. We evaluated ArchEngine with 21 junior programmers and found with strong statistical significance that it performed better than Koders (Open Hub), Krugle, and the built-in search in GitHub repository. ArchEngine performed better in both precision at 10 and Normalized Discounted Cumulative Gain (NDCG).

5.2 Future Work

There are a number of areas where we can improve our system. First, our search engine works on a limited number of tactics e.g 14, our improvement will be adding new more tactics to the search engines. Second, the current prototype shows only code snippets for the tactical file but not the neighbouring files that tactical file might interacts with, our improvement would be representing the output results such as tactical files and their dependencies in a suitable manner. Third, we have implemented a code downloader module that only downloads code projects from GitHub repository. There are many other software

repositories which we do not support yet. These repositories may require a different implementation to successfully download code projects. Therefore, another possible expansion of our project could be supporting more software repositories. Last but not the least, natural language processing (NLP) may be useful in interpreting user's intent from the query. We may consider integration of NLP to our search module to better understand user's query intent.

Bibliography

[1] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[2] 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*, pages 681–682. ACM, 2006.

[3] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4. IEEE Computer Society, 2009.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Adison Wesley, 2003.

[5] A. Blum, A. Kalai, and J. Langford. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the twelfth annual conference on Computational learning theory*, pages 203–208. ACM, 1999.

[6] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[7] P. C. Clements. *Software architecture in practice*. PhD thesis, Software Engineering Institute, 2002.

[8] D. Cutting. Apache lucene, Feb. 2005.

[9] S. G. Eick, T. L. Graves, A. F. Karr, et al. A. mockus. does code decay? assessing the evidence from change management data. In *IEEETransactions onSoftware Engineering*. Citeseer.

[10] R. Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[11] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[12] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2010.

[13] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Information processing & management*, 42(4):916–933, 2006.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[15] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[16] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., Greenwich, CT, USA, 2010.

[17] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *Software Engineering, IEEE Transactions on*, 38(5):1069–1087, 2012.

[18] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120. IEEE, 2011.

[19] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*, pages 848–858. IEEE Press, 2012.

[20] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):37, 2013.

[21] J. C.-H. Mehdi Mirakhorli. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Trans. Software Eng.*, 2015.

[22] M. Mirakhorli. Why should software architects write code?, Feb. 2016.

[23] M. Mirakhorli and J. Cleland-Huang. A decision-centric approach for tracing reliability concerns in embedded software systems. In *Proceedings of the Workshop on Embedded Software Reliability (ESR), held at ISSRE10*, 2010.

[24] M. Mirakhorli and J. Cleland-Huang. Tracing architectural concerns in high assurance systems (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 908–911. ACM, 2011.

[25] M. Mirakhorli and J. Cleland-Huang. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 123–132, Washington, DC, USA, 2011. IEEE Computer Society.

[26] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang. Archie: A tool for detecting, monitoring, and preserving architecturally significant code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 739–742. ACM, 2014.

[27] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang. Archie: A tool for detecting, monitoring, and preserving architecturally significant code. In *CM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[28] M. Mirakhorli, P. Mäder, and J. Cleland-Huang. Variability points and design pattern usage in architectural tactics. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 52. ACM, 2012.

[29] M. Mirakhorli, P. Mäder, and J. Cleland-Huang. Variability points and design pattern usage in architectural tactics. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 52:1–52:11. ACM, 2012.

[30] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 34th International Conference on Software Engineering*, pages 639–649. IEEE Press, 2012.

[31] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)*, 2012.

[32] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 639–649, Piscataway, NJ, USA, 2012. IEEE Press.

[33] O. Panchenko, H. Plattner, and A. Zeier. What do developers search for in source code and why. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 33–36. ACM, 2011.

[34] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[35] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[36] C. Seo, S.-W. Lee, and H.-J. Kim. An efficient inverted index technique for xml documents using rdbms. *Information and Software Technology*, 45(1):11–22, 2003.

[37] S. E. Sim, C. L. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 180–187. IEEE, 1998.

[38] I. University of California. The sourcerer project. sourcerer.ics.uci.edu.

[39] J. Van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):277–306, 2005.

[40] P. Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[41] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th international conference on Software engineering*, pages 513–523. ACM, 2002.

Appendix A

ArchEngine Demo

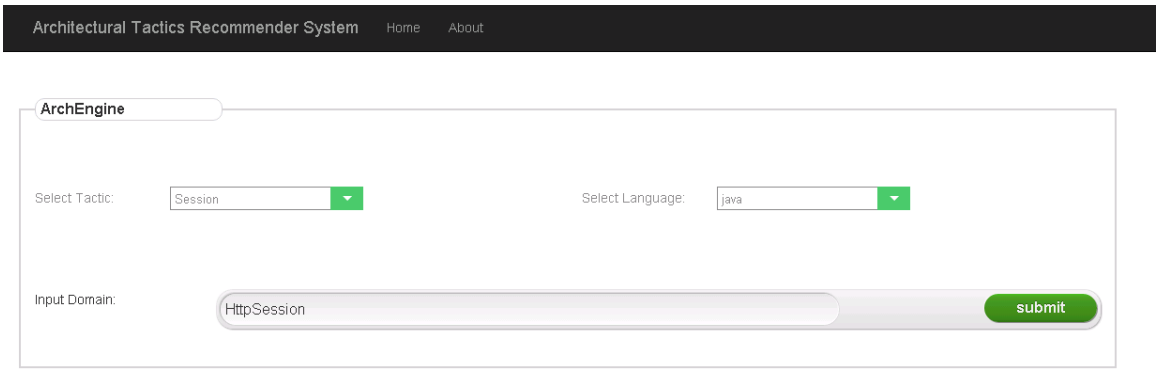


Figure A.1: Developer prompt Tactic,Language, and Technical problem to the ArchEngine

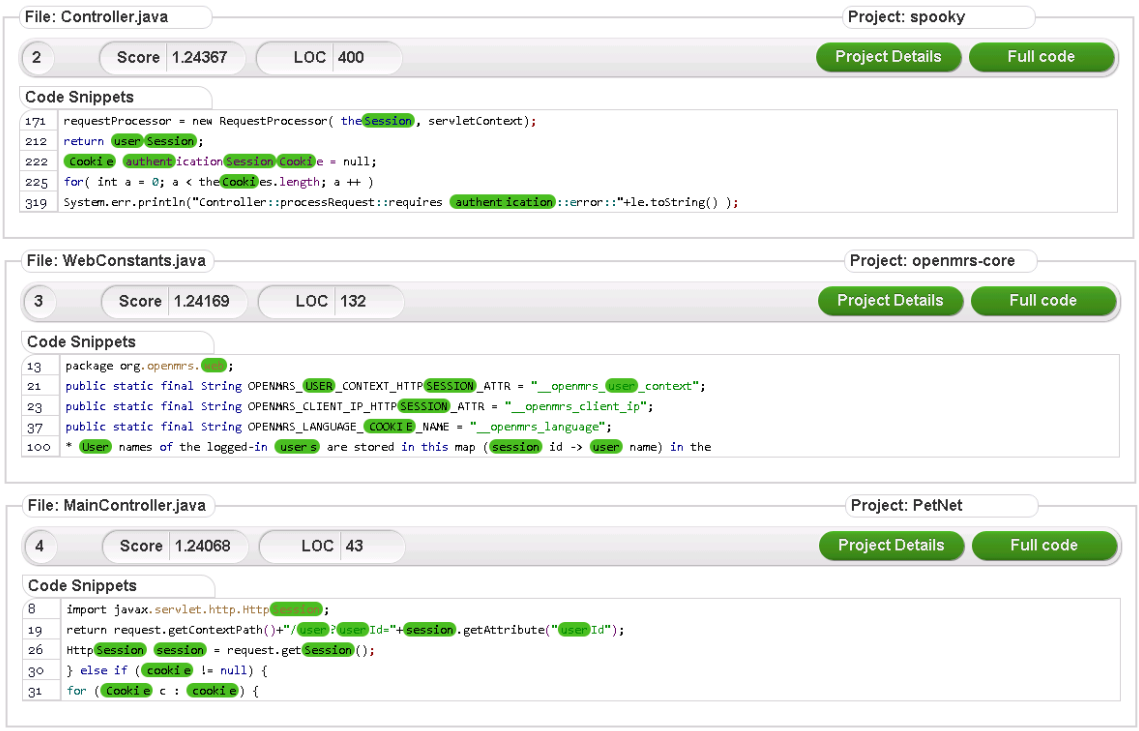


Figure A.2: A snapshot from the search result for query in figure A.1

```
public FrameworkContext getFrameworkContext( HttpServletRequest request, HttpServletResponse response, HttpSession theSession  ) throws LoginException
{
    System.err.println("Controller::getFrameworkContext");
    FrameworkContext frameworkContext = null;
    //Next we will check if the request contains an authentication
    //session id.
    Cookie[] theCookies = request.getCookies();
    Cookie authenticationSessionCookie = null;
    if( theCookies != null )
    {
        for( int a = 0; a < theCookies.length; a ++ )
        {
            if( theCookies[a].getName().equals("User_Session") )
            {
                System.err.println("Controller::getFrameworkContext::Cookie found");
                authenticationSessionCookie = theCookies[a];
                a =theCookies.length;
            }
        }
    }

    //Then we will handle the cookie or create a new security context
    if( authenticationSessionCookie == null )
    {
        System.err.println("Controller::getFrameworkContext::Cookie not found");
        FrameworkLoginContext login = new FrameworkLoginContext("Framework", new FrameworkCallbackHandler( request ) );
        System.err.println("Controller::getFrameworkContext::login.login()");
        login.login();
        frameworkContext = new FrameworkContext( login );
        System.err.println("Controller::getFrameworkContext::theSession.setAttribute()");
        theSession.setAttribute( FrameworkContext.CONTEXT_NAME, frameworkContext );
        authenticationSessionCookie = new Cookie("User_Session", "*****");
        response.addCookie( authenticationSessionCookie );
    }
    else
    {
        //We must check if the session id matches (else we must try to authenticate the user
        frameworkContext = (FrameworkContext)theSession.getAttribute( FrameworkContext.CONTEXT_NAME );
    }
    return frameworkContext;
}
```

Figure A.3: The developer click on full code button for on of the code snippets in figure A.2