A Search Engine for Finding and Reusing Architecturally Significant Code

Ibrahim Mujhid, Joanna C. S. Santos, Raghuram Gopalakrishnan, Mehdi Mirakhorli

Software Engineering Department, Rochester Institute of Technology. Rochester, NY.

Abstract

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, crowd-sourcing 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 the 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 programing analysis techniques to retrieve applications that implement these design concepts. Furthermore, it 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.

Keywords: architecture, traceability, tactics, information, models

Email addresses: `ijm9654@rit.edu` (Ibrahim Mujhid), `jds5109@rit.edu` (Joanna C. S. Santos), `rg8772@rit.edu` (Raghuram Gopalakrishnan), `mehdi@se.rit.edu` (Mehdi Mirakhorli)

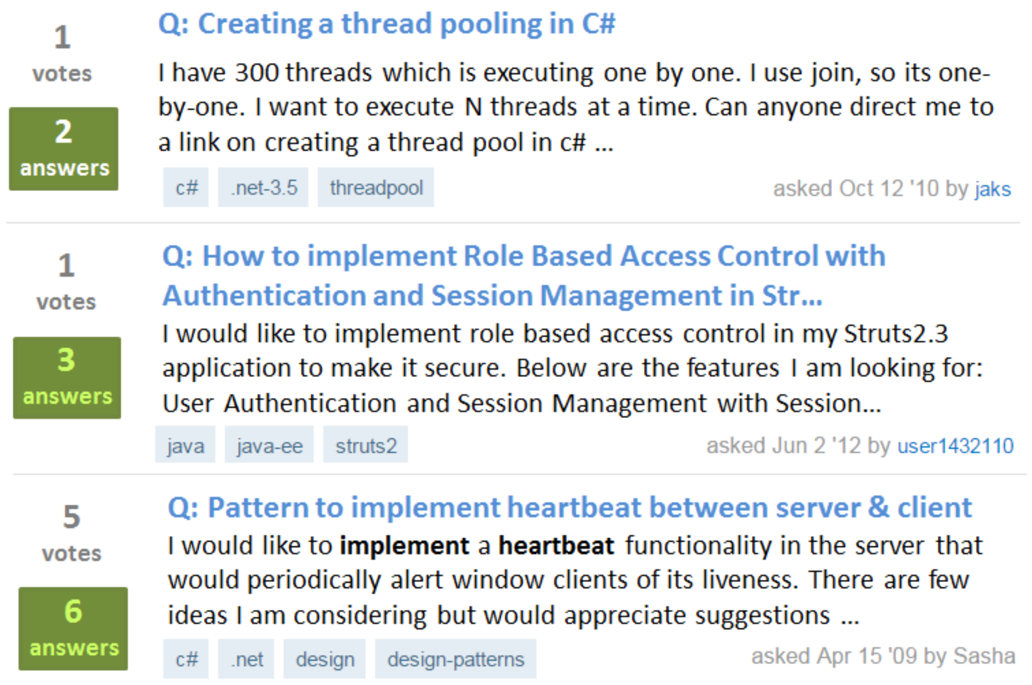
1. Introduction

In order to speed up development process, many programmers reuse existing code. Often they find that there are generic functionalities that other programmers wrote and these fragments can be reused. Socio-technical websites such as StackOverflow, and code search engines (e.g. Google Code, Koders) are the primary resources that developers often use for finding and reusing source code or even for getting ideas on how to implement a feature. However, this can be challenging when it comes to reusing *architecturally significant codes* [24] –code snippets that implement architectural patterns [8] and tactics [3]. A fundamental problem is related to the difficulties of identifying and tagging architectural patterns and tactics in the source code of a project. As a result, the current search engines such as Google Code, Koders or even those developed in academia [18] fail to incorporate these design concepts in their underlying search algorithms.

With the increasingly adoption of iterative incremental software development practices and integration of coding and design activities, there is a growing need for search engines that helps developers identify and reuse code snippets related to the architectural patterns/tactics. In a simple search through the web, one can find several examples of online posts made by developers requesting help in online forums because they did not understand how to implement specific patterns/tactics. Figure 1 shows three examples of such questions. One developer is seeking help regarding the generic implementation of a *Pooling* tactic in C#. While two others are looking for specific implementation of tactics in particular context/technology. 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 *clients and a server*.

These examples show that typically developers' 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 need to identify and index occurrence of architectural tactics, it also need to identify the **technical context** in which the tactic is implemented. State of the art, in the area of enhancing code reuse, relies on application of data-mining and natural language processing (NLP) techniques to build source code recommender systems [17], and search engines [5, 7, 28]. However, the primary intend of these techniques is retrieving generic functional code and not tactical code. Moreover, these techniques do not differentiate between the search concept and the technical context where the concept is implemented.

Difficulty of detecting architectural patterns/tactics as well as challenges for identifying the technical context within the source code of a project are the two main


1 votes

2 answers

Q: Creating a thread pooling in C#

I have 300 threads which is executing one by one. I use join, so its one-by-one. I want to execute N threads at a time. Can anyone direct me to a link on creating a thread pool in c# ...

c# .net-3.5 threadpool

asked Oct 12 '10 by jaks

1 votes

3 answers

Q: How to implement Role Based Access Control with Authentication and Session Management in Str...

I would like to implement role based access control in my Struts2.3 application to make it secure. Below are the features I am looking for: User Authentication and Session Management with Session...

java java-ee struts2

asked Jun 2 '12 by user1432110

5 votes

6 answers

Q: Pattern to implement heartbeat between server & client

I would like to **implement** a **heartbeat** functionality in the server that would periodically alert window clients of its liveness. There are few ideas I am considering but would appreciate suggestions ...

c# .net design design-patterns

asked Apr 15 '09 by Sasha

Figure 1: Developers seek help in online forums to implement architectural patterns/tactics

reasons that the notion of reusing architecturally significant source code is not well explored in the software architecture community.

In this paper, we address these limitations by presenting and rigorously validating ArchEngine, a novel search engine designed for retrieving architecturally significant code snippets. ArchEngine supports the developers in finding and reusing relevant source files that implement an architectural tactic using a particular technology.The users search request is reflected in query terms (i.e., *Authentication* using *Sprint Framework* over *HTTP* channel).

To build a source code search engine for architectural tactics, ArchEngine: (i) relies on a novel text-based classification technique to automate the discovery, extraction and indexing of architectural tactics across 116,609 open-source systems. (ii) It implements a big data compatible architecture to search efficiently through 22 million source files of these projects. (iii) It uses information retrieval [12] and structural analysis techniques [26, 13] to detect tactics and to identify the technical context in which the tactic has been used. Lastly, (iv) it utilizes 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.

In our case study, 21 professional developers evaluated the accuracy and practicality of the ArchEngine. The results show, with strong statistical significance, that users find more relevant tactical code snippets with higher precision with ArchEngine than other search engines such as SourceForge, Open Hub, Krugle and GitHub. ArchEngine is available for public use at [1].

The remainder of this paper is structured as follows. Sections 3 and 4 describe the process of mining and indexing the source code of 116,609 open source projects. Sections 5 and 6 describe detection of architectural tactics and implementation/technical context in open source projects. Sections 7 and 8 describe the ranking algorithms used to sort the results and the search process, respectively. Section 9 represent the empirical experiments that were conducted to evaluate the search engine. Section 10 describes the related work, Section 11 explains the threats to validity of this work and Section 12 summarizes the contributions of this paper and discusses future work.

2. Overview of Approach

The architecture of our search engine and its components are depicted in Figure 2. 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 represent projects and their source files in a form of index which is efficient for performing information retrieval techniques. The third component is a *tactic detector* [20, 24] which is capable of detecting 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 [20, 24].

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*. It 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.

[1] `http://design.se.rit.edu/ArchEngine/`

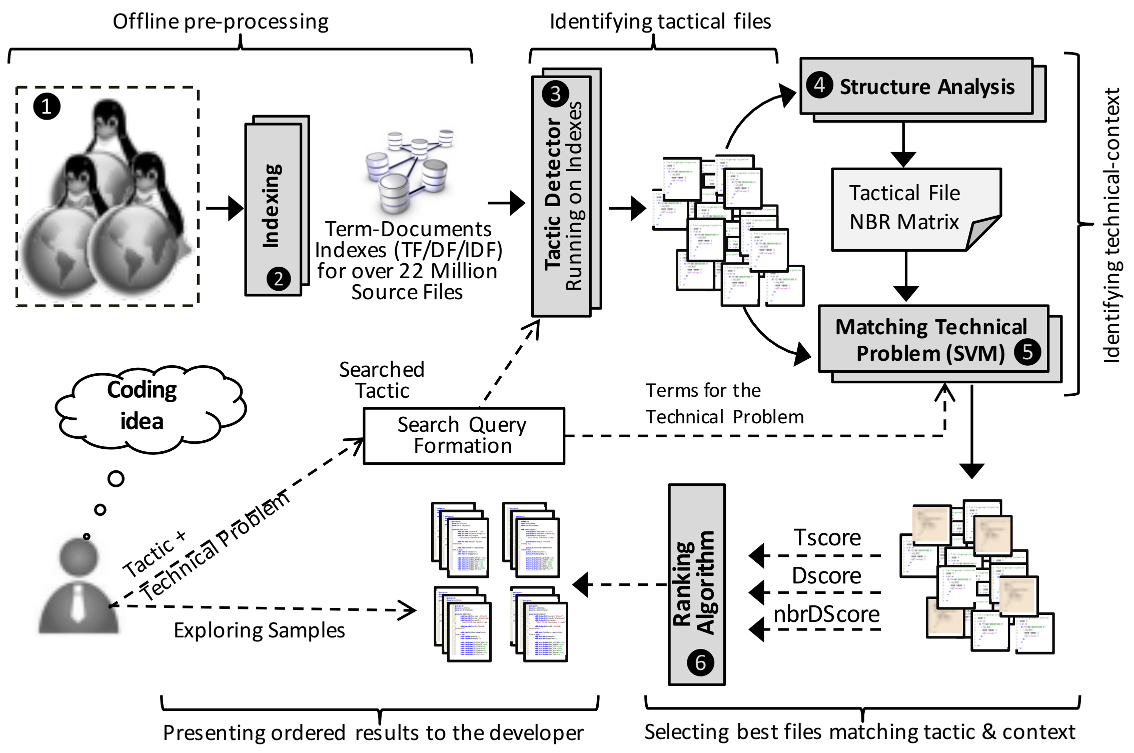

Offline pre-processing
Identifying tactical files
1
Indexing 2
Term-Documents Indexes (TF/DF/IDF) for over 22 Million Source Files
3 Tactic Detector Running on Indexes
4 Structure Analysis
Tactical File NBR Matrix
Matching Technical Problem (SVM) 5
Identifying technical-context
Coding idea
Searched Tactic
Search Query Formation
Terms for the Technical Problem
Tactic + Technical Problem
Exploring Samples
Ranking Algorithm 6
Tscore
Dscore
nbrDScore
Presenting ordered results to the developer
Selecting best files matching tactic & context

Figure 2: 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. 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 7). Subsequently, the ranking algorithm order the results based on these two metrics.

3. 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[2] 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.

We also used *Sourcerer* [29], 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 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

After 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

[2]http://ghtorrent.org/

(i.e. projects that have less than 20 source files). Table 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.

4. 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, which is called *Indexing*, first preprocesses 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 [30], (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 contain a specific term [14]. Furthermore, the index also 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.

5. 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 [25]. 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 splitted 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:

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

- *Classification Phase*: In this phase, the *indicator terms* of an architectural tactic (calculated in the *Training Phase* using the Equation 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:

$$Pr_q(c) = \frac{\sum_{t \in c \cap I_q} Pr_q(t)}{\sum_{t \in I_q} Pr_q(t)} \tag{2}$$

  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 [4], 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 [24, 20]. Currently our approach is able to accurately detect over 10 architectural tactics such as *heartbeat*, *scheduling*, *resource pooling*, *authentication*, *authorization*, *secure session management*, *ping-echo*, *checkpointing* and *audit trail*, *Role-based access control (RBAC)* [24, 23, 22, 20].

6. 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 which 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) [27] 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 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 [27], 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:

$$idf_{ti} = log_2(n/n_i) \tag{3}$$

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

$$ContextSim(f, q) = \frac{(\sum_{i=1}^{n} w_{i,f} w_{i,q})}{\left(\sqrt{\sum_{i=1}^{n} w_{i,f}} \cdot \sqrt{\sum_{i=1}^{n} w_{i,q}}\right)} \tag{4}$$

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 which 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.

7. Ranking Algorithm

In order 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 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 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 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:

$$rank(f, t, q) = TScore(f, t) + ContextSim(f, q) + \frac{\sum_{d \in nbr(f)} ContextSim(d, q)}{nbr(f)} \quad (5)$$

8. Search Process

A developer initiates the search process by first selecting the desired tactic then specifying the problem where the tactic is used to address or the technology or framework which is 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`

9. 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 [3] (which used to be known as Koders), Krugle [4], SourceForge [5] and the built-in search in GitHub repository [6]. 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, in order 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.

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.

9.1. Methodology

In this experiment, there was 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*,

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

Secure Session Management and *Thread Pooling*. A detailed discussion about these tactics is presented in [3]. Although the current version of our search engine supports 10 architectural tactics [20], for the sake of evaluating the ArchiEngine's proof of concepts, we randomely selected the above 3 tactics out of 10. 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 sample source files that they can reuse 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. Then, 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 as correct when it contained an implementation of the architectural tactic within the queried technical context or frameworks of interest.

9.2. Assigned Architecture-Prototyping Tasks

Table A.2 in the Appendix 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 task 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 subjects 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 [21].

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 which 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 alone. 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 considered 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 search engines introduced in previous subsection were used by the subjects to find sample code snippets for their architecture-prototyping tasks.

9.3. 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 [12]. 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.

9.4. 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:

$$P@k = \frac{|\{relevant\ tactical\ files\} \cap \{top\ K\ items\ in\ the\ results\}|}{k} \tag{6}$$

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.

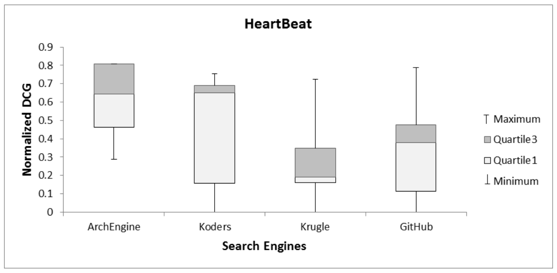
$$NDCG = \frac{rel_1 + \sum_{i=2}^{n} \frac{rel_i}{log_2(i)}}{NF} \tag{7}$$

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. 7 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).

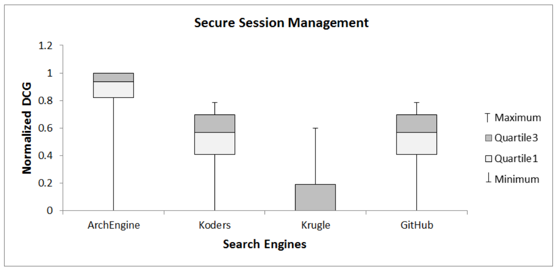
These two metrics are commonly used together to evaluate the results of web-based search engines.

9.5. Results

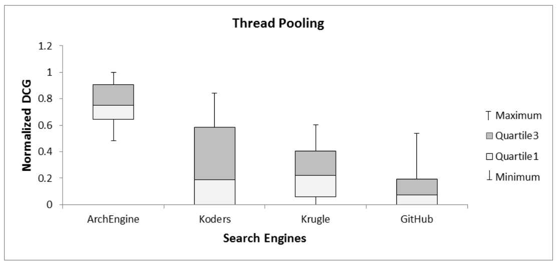
Figure 3 shows our experiment findings after evaluating the top 10 answers returned by our system and other search engines. As previously mentioned, we compared ArchEngine with Koders, Krugle, and GitHub in terms of their *P@10* and the *NDCG*. For the *P@10* metric, we calculated the mean *P@10* achieved for all 12 distinct queries evaluated by the subjects for a given tactic. For NDCG, we use a Box plot to verify how the NDCG metric is distributed for each tactic. 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.



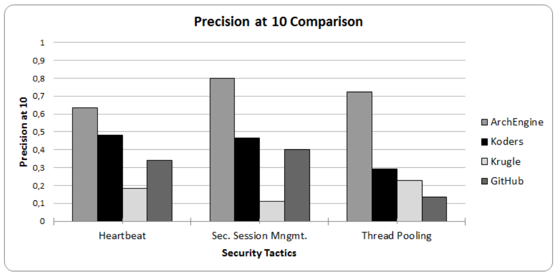
(a) NDCG results for Heartbeat



(b) NDCG results for Sec. Session Mngmt.



(c) NDCG results for Thread Pooling



(d) P@10 for all the search engines

Figure 3: Comparison of ArchEngine's performance with state-of-the-art code search engines

Regarding the P@10, 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 apearing on top of the list. Figures 3(a), 3(b) and 3(c) depicts 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, yet 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 falls into the range of 0.4 to 0.8. The results of this experiment provide a positive answer to our research hypothesis, and clearly shows the feasibility and accuracy of our search engine in extracting and returning tactic related code for a given technical problem.

10. 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 [1, 16], search queries inferred from source code [9, 31], etc), granularity level of produced output (such as functions [16], source files [1], code fragments [6, 10, 11], components [9, 31], etc), releasing approach (as a Web site [10, 15, 16], an Eclipse plugin [1, 2, 9, 11], etc), underlying code search technology and so on.

In this context, Prospector [11] 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 [11] . 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 [10] presents a code search engine which outputs Java API documentations along with sample code snippets that uses 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 [16, 19]. 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 then based on the frequency of usage (i.e. how many times a function is called). In spite of 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 it only returns C/C++ code.

Sourcerer [1, 2] 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 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 [9] and Codebroker [31] 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 generate search queries from test cases whereas Codebroker analyses the comments in the code written by programmers. Another tool for higher-level searches is Exemplar [15] which focus on finding executable Java software projects for reuse. Unlikely 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.

Besides these code search engines proposed and developed by the research community, there are also proprietary engines for retrieving specifically 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), yet their search heuristics does not emphasize finding artifacts that satisfy quality requirements through implementing architectural tactics.

11. Threats to Validity

In this section we discuss the major 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 about generalizablity of our work.

11.1. Internal Validity

Regarding our experiment design to evaluate ArchEngine, the main threat comes to 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.

Another threat is related to the accuracy of tactic detection algorithm. Since this approach relies on data mining techniques, it suffers from the common accuracy problems of automated text mining approaches. The precision of ArchEngine depends on the quality of detected tactics. Our previous extensive experiments indicate that the tactic detection achieves a precision of 70% and higher for 10 architectural tactics[20, 23, 24]. Furthermore, it is possible to tune the classification threshold of this technique to only return the source files with high confidence, increasing precision while reducing recall.

11.2. 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 subset 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 116,609 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.

12. Conclusion

We created an approach called ArchEngine for finding highly relevant source files that implement an architectural tactic within 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).

13. Acknowledgement

The work in this paper was partially funded by the US National Science Foundation grant #CCF-1543176.

[1] 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.

[2] 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.

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

[4] 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.

[5] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[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] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 475–484. IEEE, 2010.

[8] R. Hanmer. *Patterns for Fault Tolerant Software.* Wiley Series in Software Design Patterns, 2007.

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

[10] 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.

[11] 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.

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

[13] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, and J. Stage. *Object-oriented analysis & design*, volume 25. Citeseer, 2000.

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

[15] 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.

[16] 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.

[17] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 848–858, 2012.

[18] 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 Trans. Softw. Eng. Methodol.*, 22(4):37, 2013.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] R. S. Pressman. *Software engineering: a practitioner's approach.* Palgrave Macmillan, 2005.

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

[28] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 195–202. IEEE, 2006.

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

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

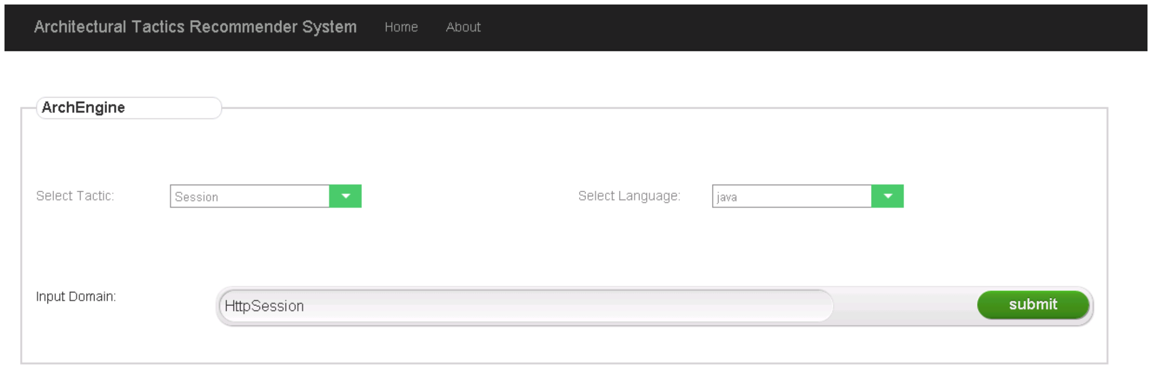
[31] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th international conference on Software engineering*, pages 513–523. ACM, 2002.

Appendix A. Tasks assigned to students in the experiment

Table A.2: Tasks assigned to students in the experiment

Task#1: Implement Heartbeat	**Task#2:** Implement Secure Session Management	**Task#3:** Implement Pooling
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.

Appendix B. ArchEngine Demo

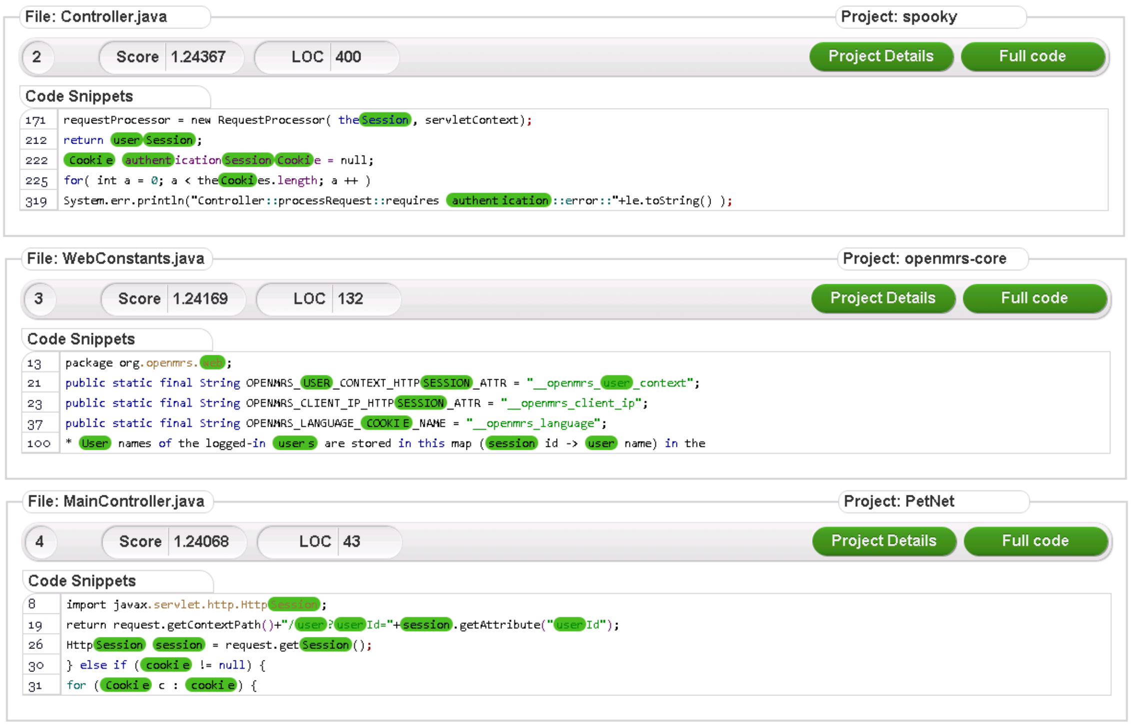


Select Tactic: Session

Select Language: java

Input Domain: HttpSession

Figure B.4: Developer prompt Tactic,Language, and Technical problem to the ArchEngine



File: Controller.java | Project: spooky | 2 | Score 1.24367 | LOC 400 | Project Details | Full code

Code Snippets

```
requestProcessor = new RequestProcessor( theSession, servletContext);
return userSession;
Cookie authenticationSessionCookie = null;
for( int a = 0; a < theCookies.length; a ++ )
System.err.println("Controller::processRequest::requires authentication::error::"+le.toString() );
```

File: WebConstants.java | Project: openmrs-core | 3 | Score 1.24169 | LOC 132 | Project Details | Full code

Code Snippets

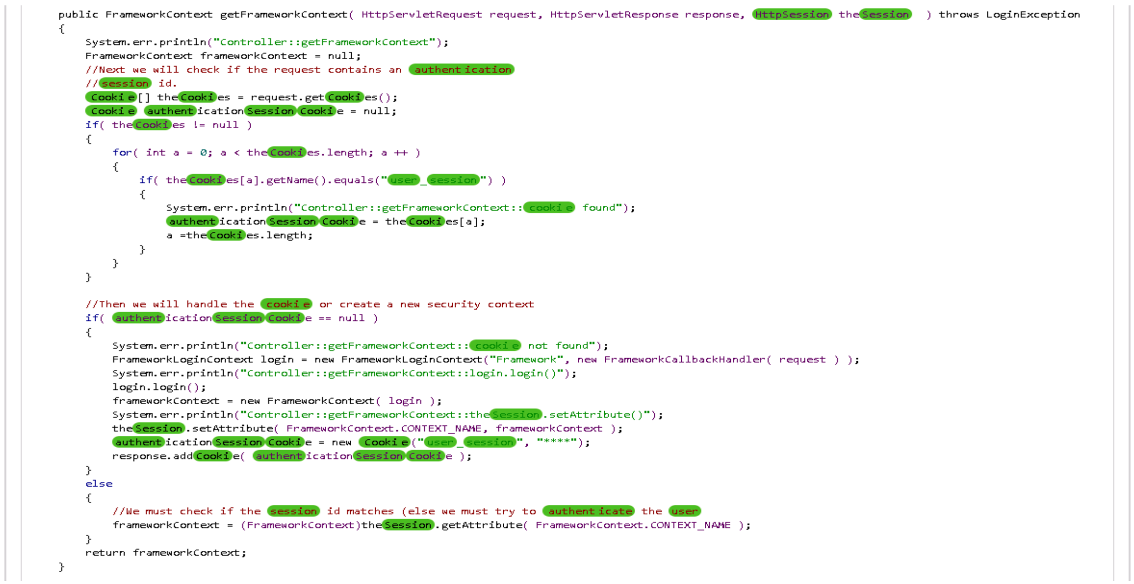
```
package org.openmrs.web;
public static final String OPENMRS_USER_CONTEXT_HTTPSESSION_ATTR = "__openmrs_user_context";
public static final String OPENMRS_CLIENT_IP_HTTPSESSION_ATTR = "__openmrs_client_ip";
public static final String OPENMRS_LANGUAGE_COOKIE_NAME = "__openmrs_language";
* User names of the logged-in users are stored in this map (session id -> user name) in the
```

File: MainController.java | Project: PetNet | 4 | Score 1.24068 | LOC 43 | Project Details | Full code

Code Snippets

```
import javax.servlet.http.HttpSession;
return request.getContextPath()+"/user?userId="+session.getAttribute("userId");
HttpSession session = request.getSession();
} else if (cookie != null) {
for (Cookie c : cookie) {
```

Figure B.5: A snapshot from the search result for query in figure B.4



```
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 B.6: The developer click on full code button for on of the code snippets in figure B.5