Thesaurus-Based Automatic Query Expansion for Interface-Driven Code Search

Otávio A. L. Lemos, Adriano C. de Paula, Felipe C. Zanichelli
Department of Science and Technology
Federal University of São Paulo
S. J. dos Campos, SP, Brazil
{otavio.lemos; adriano.carvalho; felipe.zanichelli}@unifesp.br

Cristina V. Lopes
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA, USA
lopes@ics.uci.edu

ABSTRACT

Software engineers often resort to code search practices to support software maintenance and evolution tasks, in particular code reuse. An issue that affects code search is the vocabulary mismatch problem: while searching for a particular function, users have to guess the exact words that were chosen by original developers to name code entities. In this paper we present an automatic query expansion (AQE) approach that uses word relations to increase the chances of finding relevant code. The approach is applied on top of Test-Driven Code Search (TDCS), a promising code retrieval technique that uses test cases as inputs to formulate the search query, but can also be used with other techniques that handle interface definitions to produce queries (interface-driven code search). Since these techniques rely on keywords and types, the vocabulary mismatch problem is also relevant. AQE is carried out by leveraging WordNet, a type thesaurus for expanding types, and another thesaurus containing only software-related word relations. Our approach is general but was specifically designed for non-native English speakers, who are frequently unaware of the most common terms used to name functions in software. Our evaluation with 36 non-native subjects – including developers and senior Computer Science students – provides evidence that our approach can improve the chances of finding relevant functions by 41% (recall improvement of 30%, on average), without hurting precision.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Theory, Algorithms, Experimentation

Keywords

software reuse, code search, automatic query expansion

MSR '14, May 31 - June 1, 2014, Hyderabad, India

1. INTRODUCTION

A recently popular form of software reuse relies on querying over and copying open source code available on the Web. Such approach has been identified by researchers as the new facet of software reuse [23, 40, 41].

However, finding and reusing code available on the Web is not straightforward: searches are frequently based solely on text, dependencies have to be manually extracted, and pieces of code have to be copied and woven into the workspace by hand [19]. Moreover, a general Information Retrieval (IR) problem that affects code search is the vocabulary mismatch problem [3], which refers to the gap between the names used to describe concepts in queries and the names used by authors to describe the same concepts [45].

Test-Driven Code Search (TDCS) is a promising approach to code search that uses test cases as inputs to formulate the search query. A recent study shows that reuse through an implementation of TDCS can be 66% faster than a general-purpose code search engine, and three times faster than implementing functions manually[1] [19]. In TDCS, the interface of the desired function – including return type, parameters types and method names – is extracted from the tests to form a specialized query (what we call here *interface-driven* code search). Then, matched code candidates can be automatically sliced out with all their dependencies, woven into the workspace, and ran against the test cases to verify suitability in the local context.

Although TDCS has been shown to be effective specially in the context of auxiliary functions [19], it still relies on keywords to match functions, so the vocabulary mismatch problem also affects its performance. One of the most natural and successful techniques to deal with the vocabulary mismatch problem is known as *automatic query expansion* (AQE). The basic idea is to expand the original query with other words that best capture the user intent, or that simply produce a query more likely to retrieve relevant documents [7]. Use of AQE generally increases recall, and is widely used in diverse science and engineering fields [26]. Albeit there has been recent effort in the software engineering community to try out query refinement techniques, most of them tend to place additional burden on the developer [39] – *i.e.*, are not fully automatic –, or do not target the specificities of interface-driven code search (*e.g.*, do not handle types).

In this paper we propose an AQE approach for interface-driven code search that takes advantage of the generally high precision given by test case and interface definition used to

[1]Similar test-driven code search approaches were proposed by other researchers [16, 33].

formulate queries in TDCS [15]. Our approach makes use of synonyms to improve the chances of finding relevant code and the overall recall of the searches. To generate the synonyms for the interface-driven query terms, we adopt three thesauri: (1) WordNet [27] and (2) a code-related thesaurus built by using Yang and Tan approach [46] (to cover words that are semantically related in software but not in English [46]) – for the natural language terms –; and (3) a type thesaurus to expand the types contained in the interface-driven queries (*i.e.*, return and parameter types). The two natural language thesauri are also used to expand queries with antonyms, to improve precision (*e.g.*, to prevent that *unzip* is matched while searching for *zip*).

The approach presented in this paper is general, but was specially designed for users that have English as a second language. We noticed these users are frequently unaware of the most common terms used to name functions, and so usually produce queries with very low recall. We argue that these users many times select *close-to-adequate* terms, which with the help of synonyms can produce better recall. Given that context, to evaluate our AQE approach, we have conducted a controlled experiment involving 36 non-native English speaking subjects – among them professional developers and senior Computer Science students –, and realistic queries given by these subjects to find 13 different functions in an unbiased code repository. We compared the performance of the queries in terms of their ability to find relevant code and also in terms of recall, in three different scenarios: (1) with WordNet only; (2) with the code thesaurus only; (3) with the type thesaurus only; and (3) with the three thesauri.

Our study shows that our AQE approach can improve the chances of finding relevant code by 41% (*i.e.*, the approach improved the number of successful queries by such proportion), and recall by 30%, on average, a difference statistically significant at 95% confidence level. The best results were achieved when the three thesauri were applied. Another remarkable result is that our approach significantly improved the chances of finding relevant code and recall even when only WordNet was used (recall was improved by 20%, on average). This is quite surprising, as some authors have argued that natural language thesauri are generally not appropriate for code search and software engineering in general [11, 42, 46]. We believe WordNet is effective here because of our context, which involves non-native English speakers and interface-driven code search. In this type of search, the search space is first narrowed down by using interface definitions, and then broadened in a controlled manner with the use of query expansion, to match a larger number of relevant functions.

The remainder of this paper is structured as follows. Section 2 provides background to understand TDCS and our target implementation, and Section 3 presents our AQE approach with its implementation. Section 4 presents the study setup for our evaluation, and Section 5 shows the results of our experiment. Section 7 provides a discussion of work related to ours and Section 8 concludes the paper.

2. BACKGROUND

Since the end of the 1990s – and probably before that – one of the most common practices of software engineers has been to search for source code [38]. In this endeavor developers often resort to general purpose search engines like Google, providing keywords to match the implementations they desire. Specialized code search engines like ohloh Code[2] have improved the chances of finding relevant code by adding filtering (*e.g.*, by programming language) and program syntax (*e.g.*, search within method names only). However, these search engines are still tied to keywords and syntactic information, which might cause missing of candidates that implement the desired behavior but do not match the structural information contained in the query, or the opposite. In an attempt to deal with such shortcoming, more recently some approaches have added semantic information to the search, improving the chances of finding relevant code.

2.1 Test- and Interface-Driven Code Search

Test-Driven Code Search (TDCS) is one of the semantics-based code search approaches that were recently proposed to improve code search [19]. TDCS makes use of test cases to describe a desired feature and check the suitability of code candidates in the local context.

In TDCS, before the search, test cases must be written for the entry point method of the desired function. For instance, suppose a developer is looking for the implementation of a function that reverses a text string. The test case <("*reverse*"), "*esrever*"> in the form <($I_1, ..., I_n$), EO>, where EO is the expected output of the desired function when $I_1, ..., I_n$ is used as the set of inputs, could be used to verify whether the function behaves correctly for a sample string. After writing the test case, the user can trigger the search, which extracts the desired function interface, and look for potential matches in the repository. After returning the matched candidates, the function is sliced out from the matching project with whatever structure it depends upon, in order to form a compilable unit. This unit is then woven into the user's workspace and ran against the test cases. If the test cases run successfully on the potential match, the function can be left woven, being reused in the local project[3].

Note that there are two types of code search involved in this process: (1) *interface-driven code search*, when code candidates that possess an interface similar to the one designed for the desired function are searched for (*e.g.*, in the example, functions with the interface `string f(string)`[4]); and (2) *test-driven code search*, when test cases are run against matched candidates to verify their suitability in the local context.

Interface- and Test-Driven Code Search share many similarities with other search approaches proposed in the past. For instance, *signature matching* is similar to *interface-driven*

[2] `http://code.ohloh.net` - 08/22/2013.

[3] Note that this works not only for static methods, but also for instance methods [20]. That is, the test cases can target an entry point method that accesses fields of the class it belongs too. In that case the slicer would pull out the required fields in order to form a compilable unit. Moreover, if the method depends on other classes/methods, these are also pulled into the workspace, so that the entry point method can be compiled. *Merging* of classes *by name* as done in Hyper/J is also performed when the function targeted by the test cases belong to an existing class [30].

[4] In our current implementation, we only consider return type, method name, and parameter types to form an interface. Other elements could be handled, such as field types of the class the method belongs to, thrown exceptions, etc. However, we believe the information we use is adequate for the proposes at hand. By using other elements in the interface, the search space might become too narrow, resulting in poor recall. However, it must be clear that sometimes the test case query needs to be adapted to conform to the other elements of the interface not dealt on the search. Some of these adaptations can be done automatically. In fact, Reiss's TDCS approach [33] incorporates some types of transformations that target such problem.

search in the context of code libraries. Moreover, similar TDCS approaches were implemented by other researchers. We summarize these approaches in Section 7.

An important characteristic of TDCS is that it tends to improve the precision of code search because if a matched candidate ever passes on the designed test cases it is very likely to implement the desired function. In fact, in a TDCS study presented before, all candidates that passed the test cases did implement the intended functions, raising precision to 100% [19]. Hummel and Janjic [15], one of the first to explore TDCS, also argue that it is a key approach to improve the precision of search engines for software reuse. In fact, interface-driven code search itself generally improves precision, since when an interface is specified for the desired function, the search space is largely reduced. For instance, with the target repository used in the experiment described in Section 5, when restricting the return type of a function to `int`, the number of returned results go from 2,001,648 to 145,488, a reduction of 92%.

On the other hand, restricting the interface of the desired function may also likely cause poor recall, since it may be difficult to find the desired function with the originally designed interface, while matching the natural language terms. The approach presented in this paper handles this problem, since expanding the query generally enlarges the search space.

2.2 TDCS Java Implementation: CodeGenie

CodeGenie, the implementation we target in this paper, is an Eclipse plugin that uses the facilities included in this platform to perform TDCS. The tool relies on Sourcerer [24], a source code infrastructure that provides all support needed to perform TDCS. The current implementation of Sourcerer receives Solr queries to execute the interface-driven searches.

To search in CodeGenie, users first have to develop JUnit test cases for the desired function. The tool then extracts the interface of the desired function's entry point method, and formulates queries that contain three parts: (1) return type of the method; (2) keywords present in the entry point method name; and (3) parameter types of the method. For example, given a JUnit implementation of the test case mentioned above, which uses the assertion `assertEquals("esrever", Util.invert("reverse"))`, CodeGenie formulates a Solr[5] query similar to the following:

return_type:(String)
method_name:(invert)
parameters_types:(String)

The query above means: "look for an entry point method of a function that contains the term 'invert' in its name, returns a String, and receives a String as a parameter"[6]. Sourcerer then returns matching candidates that can be woven into the workspace and ran against the test cases. The current CodeGenie implementation automatically weaves and tests candidates and shows results to the users based on the runs. The user can then choose which successful candidate to reuse in the local project.

2.3 Query Refinement

According to Carpineto and Romano [7], the relative ineffectiveness of IR systems is largely caused by the inaccuracy with which a query formed by a few keywords models the actual user information need. Another known issue is that, in most collections, the same concept may be referred to using different words. This problem, known as *synonymy* (or vocabulary mismatch problem), has an impact on the recall of most IR systems [26]. This is also true in the context of code search, since keywords are still required.

Query refinement (QR) methods have been proposed to deal with this issue. In general, these methods are classified as *global* or *local*. Global methods expand query terms independent of the results returned from the original query, so that changes in its wording will cause the new query to match other semantically similar terms. Local methods, on the other hand, adjust a query relative to the documents that initially appear to match the original query [26].

Automatic query expansion (AQE) is a well known global method to overcome the vocabulary mismatch problem [7]. It augments the user's original query with new features with a similar meaning. In the case of the technique proposed in this paper, we decided for a global approach that does not require running the query prior to expanding it: the query is first expanded with related terms, and then executed to find relevant functions. This strategy generally improves performance, as we only need to run the query once. Another advantage is not requiring any additional user input [26].

3. AUTOMATIC QUERY EXPANSION FOR INTERFACE-DRIVEN CODE SEARCH

We believe that AQE has not yet been fully explored in the context of code search and, to the best of our knowledge, no researchers have yet investigated its use within interface-driven code search and TDCS as described in the previous section. AQE is appropriate for our context because, as discussed in the previous section, these approaches generally perform well with respect to precision, but often compensated by the corresponding loss in recall.

In the previous section, we showed how interface-driven code search is performed in CodeGenie, and how queries are formulated based on a set of test cases. The interface-driven query has three parts: *return type*, *method name*, and *parameters types*. Query expansion can be applied to all parts, but there is a clear difference between the *method name* part of the query, which includes *natural language terms*, and the other parts, which are basically formed by *types*. Next, we describe our approach to query expansion in terms of each of these categories.

3.1 Expanding Natural Language Terms

The second part of our query is composed of English and code-related keywords that describe the desired function. For instance, `getName` would generate the terms `get` and `name`; `zipFile` would generate the terms `zip` and `file`. Code candidates must contain these keywords in their method names in order to be matched.

It is important to note that this part of the query is very limited, *i.e.*, it is composed of a few terms (frequently two or three), since method names in Java are usually not large. Therefore, if these names are poorly chosen by the user, due to the impact of the vocabulary mismatch problem, the query will bring few or no results at all. We believe that this is particularly true for non-native English speakers, who are frequently unaware of the most common terms used to define specific functions in code, in English. We hypothesize that these users choose words that are *close to adequate*, when they have at least a basic level of English knowledge.

A typical example appears in the context of the *reverse string* function, mentioned in Section 2. For a Portuguese

[5] `http://lucene.apache.org/solr/` - 02/05/2014.

[6] A camel case splitting algorithm is used to extract the terms used in the *name* part of the query, since this is a standard Java naming convention.

native, the term *invert* makes much more sense than *reverse* in this case, because the Portuguese word *inverter* is the one used with the intended meaning in that language, while *reverter* is more frequently used to mean *revert* in the sense of *going back to a previous state*. As Portuguese is a Latin-based language, the closest word in English more natural to be chosen is the likewise Latin-based "invert". However, if that word is used, the user would probably not find any matches, since the most adequate term is *revert*. To improve the chances of finding relevant code, we could expand the initial queries with synonyms indicated by an English thesaurus, without requiring user intervention (hence the *automatic* approach). In the abovementioned case, the thesaurus would indicate that *reverse* is an *invert* synonym, making the query match results for both keywords.

Antonyms can be used to filter out undesired results. For instance, while searching for a function containing the term `zip`, functions containing the term `unzip` would also be matched. Thus, in our approach, antonyms are also used to improve precision.

Our current implementation uses WordNet to generate English synonyms and antonyms, and a code-related term thesaurus (called henceforth *code thesaurus*) with software-related word relations. Such thesaurus was populated with terms generated by the application of Yang and Tan's [46] approach to identify word relations in code. The researchers have fed their tool with some Java libraries (in particular, Apache's Commons and Google's Guava), and returned the results to us. We then filtered out the results manually, and populated our thesaurus. Our thesaurus also contain other common software word relations and gold sets that were used in other studies (such as in [42][7]).

3.2 Expanding Types

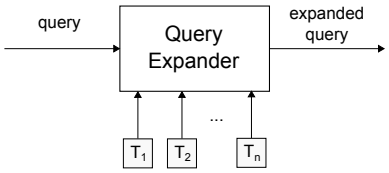
Another instance of the vocabulary mismatch problem that might occur in the context of interface-driven code search is related to the types used in the query. For instance, suppose that a user is looking for an *image rotation* function and develops a query for a method that returns `void` and receives two parameters: (1) an `Image`; and (2) an `Integer` rotation degree parameter. If there was a method in the repository that implements the desired function but instead of receiving the `Integer` parameter, received an `int`, the method would not be matched.

Therefore, to increase the chances of finding functions that implement interfaces similar to the ones used for the query, we also use a type thesaurus to expand type terms. This is important because, as discussed in Section 2, defining the interface of the desired function in the query restricts the search space in a large degree.

Our initial type thesaurus does simple numerical and Java collection types expansion. For instance, if the user defines a query for a method that returns an `Integer`, we also look for methods that return `int`.

3.3 Query Expander

The basic idea of our approach is to generate expansion features for each of the terms available on each part of the query, and add those terms to the original query (*i.e.*, we generate *one-to-one* associations [7]). This is done for each of the thesauri configured to be used in the query expander. Our approach is general in the sense that several thesauri can be used, and it supports the configuration of which are going to be applied to which query part. Figure 1 shows a diagram that depicts our query expansion approach. The Query Expander module receives a *query* as an input and generates an *expanded query* according to which thesauri is configured to be used with it.



Legend: T - Thesaurus.

Figure 1: Our query expansion approach.

The main idea is to give the same weight to the original terms and their synonyms so that the query will match candidates that have at least one keyword related to each of the concepts initially described by the user. For instance, if the query terms are *get* and *name*, we want to match candidates that have either *get OR* one of its synonyms, *AND name OR* one of its synonyms. For the antonyms, we do not want to match candidates that contain any of them. This is consonant with the *revert string* example discussed before, with respect to the support for non-native speakers, and it also prevents the query from matching too many candidates, were a disjunction to be used among terms.

More formally, let $(t_1 \ldots t_n)$ be the n terms of a query part; s_{ij} the j-th synonym of the i-th term; a_{ij} the j-th antonym of the i-th term; m_i the number of synonyms generated for the i-th term; and o_i the number of antonyms generated for the i-th term. For each thesaurus, our approach generates a query that must match terms according to the following logic:

$$(t_1 \vee s_{11} \vee \ldots \vee s_{1m_1}) \wedge \ldots \wedge (t_n \vee s_{n1} \vee \ldots \vee s_{nm_n}) \wedge \neg(a_{11} \vee \ldots \vee a_{1o_1}) \wedge \ldots \wedge \neg(a_{n1} \vee \ldots \vee a_{no_n})$$

3.4 Implementation: CodeGenie 2.0

Our current implementation uses WordNet and a code-related terms thesaurus to expand terms in the method name part of the query, and a type thesaurus to expand types in the return type and parameter types parts of the query. Other thesauri could be used. For instance, a domain-specific thesaurus could be used in the context of a company that implements code in that domain (*e.g.*, with health care terms).

We implemented the automatic query expansion approach on top of CodeGenie[8]. However, since the query expansion is done by means of a separate implementation service, it could also be configured to be used with any other interface-driven code search tool that generates queries using the format described in Section 2.

4. STUDY SETUP

Our goal is to investigate the impact of using English, code, and type thesaurus query expansion in the context of interface-driven code search. When using this type of approach, we are looking into broadening our search space to find more relevant functions. As discussed in the previous section, this may be of particular interest to non-native

[7]Obtained through personal communication with Emily Hill and Lori Pollock.

[8]CodeGenie 2.0 is not yet available for download as it is still under test. However, we have set up a Sourcerer web interface so that users can perform interface-driven code search on our test repository (see Section 4). Such service can be accessed at `http://192.241.180.208:8080/sourcerer-idcs/`

English speakers. Therefore, we evaluate the approach presented in this paper in the context of non-native English users (specifically Brazilian Portuguese native speakers), for the different configurations that the approach can assume when using the different adopted thesauri. In this endeavor, we are interested in the following research questions:

In the context of interface-driven code search for non-native English speakers:

RQ1. Does query expansion based on WordNet improve recall?

RQ2. Does query expansion based on code thesaurus improve recall?

RQ3. Does query expansion based on type thesaurus improve recall?

RQ4. Does query expansion based on the three thesauri (WordNet, code, and type) improve recall?

Our investigation develops in terms of four hypotheses derived from these research questions. The null (0) and alternative (A) definitions of each hypothesis are described in Table 1. Note that we have one hypothesis for each research question: hypothesis H_i is related to research question R_i.

Table 1: Hypotheses formulated for our experiment.

	Null hypothesis (0)	Alternative Hypothesis (A)
H_1	$Recall_{woQE} = Recall_{QE_w}$	$Recall_{QE_w} > Recall_{woQE}$
H_2	$Recall_{woQE} = Recall_{QE_c}$	$Recall_{QE_c} > Recall_{woQE}$
H_3	$Recall_{woQE} = Recall_{QE_t}$	$Recall_{QE_t} > Recall_{woQE}$
H_4	$Recall_{woQE} = Recall_{QE_{wct}}$	$Recall_{QE_{wct}} > Recall_{woQE}$

Legend: H = Hypothesis, woQE = without query expansion, QE_w = WordNet query expansion, QE_c = code thesaurus query expansion, QE_t = type thesaurus query expansion, QE_{wct}= query expansion using the three thesauri.

4.1 Repository, Functions, and Queries

Repository. According to Fraser and Arcuri [8], several authors that propose novel test data generation techniques do not conduct sound empirical studies to provide evidence for the usefulness of their approaches. This happens because case studies tend to be either small or biased toward a specific kind of software. To cope with this problem, the authors have randomly selected 100 Java projects from SourceForge, in what was probably the first attempt to run a statistically sound experiment in software testing. The resulting benchmark, called SF100, is statistically sound and representative for open source projects[9]. We feel this issue also affects the evaluation of code search techniques: many times the target repositories are not a random sample of software projects, but a specific set of projects, which could introduce bias to the study. Concerned with this problem, and to construct a manageable and unbiased repository for our experiment, we used Sourcerer to index the same set of Java projects used by Fraser and Arcuri [8].

Functions. For our experiment, we have selected a set of functions to be searched in the repository. The idea was to simulate a situation where the user would look for an implementation of a specific function. To avoid bias, we looked into functions that were used in previous code search studies [12, 19, 33].

To have a list of realistic features from that first set, we selected only those that appeared at least twice in our target repository. In this way we could be assured that these functions were in fact implemented and used in real software projects. The functions were manually searched in the repository, by using several keywords and text-based matching, and careful inspection of hits. This procedure was important to enable measuring the evaluation metrics recall and precision afterwards.

[9] http://www.evosuite.org/sf100/ - 08/08/13

Table 2: Functions selected for our study.

#	Description	Freq.
1	Computing the MD5 hash of a string	3
2	Decoding a URL	3
3	Encoding Java strings for HTML displaying	9
4	Encrypting a password	6
5	Filtering folder contents with specific file types	5
6	Generating the complementary DNA seq.	2
7	Generating the reverse complementary DNA seq.	2
8	Joining a list of strings in a single string	6
9	Revert a text string	2
10	Rotating an image	2
11	Saving an image in JPG format	3
12	Scaling an image	6
13	Zipping files	4

The features are *auxiliary functions*, *i.e.,* supportive actions of software systems that comprise 10-200 lines of code [22]. According to a recent study that analyzed a large log of queries submitted to a code search engine [2], this type of function (called there *programming tasks*) is the one most frequently targeted by developers. Another recent study showed the importance of auxiliary functions to software development in general [22]. Therefore, we believe these features are adequate for the purposes at hand. The functions selected for our study together with the frequency in which they appear in the target repository are listed in alphabetical order in Table 2.

Queries. To adequately evaluate our approach, we had to look into queries that users would really use while performing interface-driven code search; that is, we could not guess which interfaces they would pick without biasing our study. Therefore, we have conducted a survey to collect realistic interface definitions that users would use in their queries. Users were presented with the functions' descriptions in their native language – in our case, Portuguese –, and were required to define what interfaces they would use to search for those functions in English. Since we are interested in interface-driven code search, we asked them to guess the return type, name, and parameter types of the entry point method of the function that would implement the desired behavior. Then, to measure precision and recall for those queries, we conducted the searches by using the interfaces defined by the users.

Among other profile questions, our survey included questions about the English knowledge level of the participant, and whether they were students or professional developers. 36 subjects responded to our survey; from them, 24 were senior Computer Science undergraduate students and 12 were developers; 14 had advanced English knowledge, and 22 had basic English knowledge. Each subject guessed interfaces for each of the 13 target functions of our experiment, summarizing 468 queries.

4.2 Statistical Analsysis and Experimental Design and Procedure

For the conducted experiments, we adopted the *repeated measures* experimental design, where each query was evaluated before and after applying the expansions. Such type of

design supports more control to the variability among the subjects (in this case, the queries) [29].

From a statistical standpoint, a simple observation of the means from sample observations is not enough to infer about the actual populations. This happens because the reached differences might be a coincidence caused by random sampling. To check whether the observed differences are in fact significant, statistical hypothesis tests can be applied.

In our study, each query was used to search for a function with and without the application of query expansion. In this case, the *paired* statistical hypothesis tests can compare measures within items rather than across them. Paired tests are considered to greatly improve precision when compared to unpaired tests [29]. Before choosing the more adequate test to be applied, we must verify the normality of our observations. A Shapiro-Wilk test indicated that the observations in our experiments did not follow a normal distribution. Therefore, we decided to apply the Wilcoxon/Mann-Whitney non-parametric signed-rank paired test, which does not assume normal distributions [36].

Significance levels of 0.01, 0.05 and 0.1 have been used for statistical tests in Software Engineering [5, 28]. Although some of our tests were significant at 0.05, we decided to adopt a significance level of 0.1, mainly motivated by the size of our target repository, which was not very large. We believe that a sensible difference at 90% confidence level is considerable given our scenario. Other experiments in the past have also adopted such threshold [6, 18, 47]. Thus, our analyses consider p-values below 0.1 significant. For all statistical tests we used the R language and environment[10].

4.3 Metrics

To evaluate the performance of the queries while applying the proposed query expansion approach, we adopted some metrics commonly used to evaluate information retrieval systems. We measured the *number of successful queries* (#SQ), which refers to the number of queries that retrieved at least a single relevant function for the experiment; the *number of relevant functions retrieved* for the experiment (#RFR); and *recall*. #SQ and #RFR are important in our context because returning a single relevant function can already be considered a success; *i.e.*, we are more interested in reusing code than in matching all or most relevant functions available in the repository.

Although our hypotheses do not target *precision*, since it is reasonable to expect that it will be raised to 100% when the test cases are run in TDCS, we have also included this metric to our experiment (it is measured before running the test cases, to check the amount of spurious results). The idea is that we do not want the performance of TDCS to be worsen when the AQE approach is applied (the more the noise results are returned, the more test cases need to be run, affecting TDCS's performance). Since we are adding terms to the query, we could hurt precision.

In our context, recall is defined as the intersection between the number of retrieved functions and the number of relevant functions (that is, functions in the repository that implement the desired feature and contain an interface similar to the one defined by the user[11]), over the number of relevant functions; and precision is defined as the intersection between the number of retrieved functions and the number of relevant functions, over the number of retrieved functions. Recall is a measure of completeness or quantity, whereas precision is a measure of exactness or quality.

Recall and precision, as used in our context, are formally defined as follows:

$$\textbf{recall} = \frac{|\text{retrieved functions} \cap \text{relevant functions}|}{|\text{relevant functions}|}$$

$$\textbf{precision} = \frac{|\text{retrieved functions} \cap \text{relevant functions}|}{|\text{retrieved functions}|}$$

We have set an upper bound to the result set size while performing the searches to 100 (for ranking we rely on Solr's algorithms). We believe this is a reasonable boundary since it is not realistic to think that a user would go over more than such amount of code candidates while looking for a function. Moreover, we also believe this is an adequate number of results that could be automatically tested by using TDCS. A result set larger than that could significantly reduce the performance of this approach. Also, for the recall and precision measurements, we only considered the queries that match at least a single relevant result by using one of the approaches (without applying any expansion, applying only WordNet, applying only the code thesaurus, applying only the type thesaurus, and applying the three thesauri). This is quite reasonable as we are not measuring how good users are in defining queries, but how the approach can help when a reasonable query is designed.

5. RESULTS AND ANALYSIS

Table 3 shows the main results of our experiment. In particular, we show descriptive statistics for the measured recall when each approach is applied (mean, minimum, maximum, and standard deviation), and the number of successful queries (#SQ) and relevant functions retrieved (#RFR). To have a general view of the outcomes, the graph presented in Figure 2 summarizes our results in terms of the number of relevant functions retrieved by each approach. We can clearly see that the AQE approach presented an improvement with respect to this metric, and increasingly according to the thesauri used (except for the code thesaurus that did not show any significant benefit in this experiment).

Note that recall, the number of successful queries, and the number of relevant functions retrieved are improved when our query expansion approach is applied. In particular, the number of successful queries improved by 41% when the composed approach was applied. Other noteworthy results not shown in the table are the following: 20 out of the 36 subjects (55%) attained better results when the composed query expansion approach was applied to their queries (for the rest, results remained the same after applying the expansion); 7 out of the 36 subjects (19%) only found relevant functions when a query expansion approach was applied; and 2 out of the 13 target functions (15%) were only found when one of the query expansion approaches was applied. Next we analyze our results in face of the research questions presented in the previous section.

WordNet query expansion. First, we analyze the results when the WordNet query expansion approach was used,

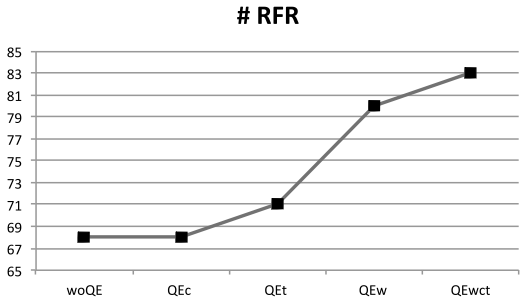
[10] `http://www.r-project.org/` - 08/05/13

[11] We define interface similarity in terms of return type and number of parameters. That is, if the query defined by a user targets a function that returns some value (*i.e.*, not `void`), relevant functions in the repository that return `void` are not considered, and vice-versa. Also, relevant functions in the repository that possess a different number of parameters than the function defined in the query are not considered either. We believe this is fair since, for instance, a user looking for a function that returns an `int` is not interested in functions that return `void`, the same way a user looking for a PDF document in Google (by using the *filetype:PDF* feature in the query) is not interested in `.DOC` documents.

Table 3: Results for the main metrics used in our experiment, for each approach.

Approach	Recall				#SQ	#RFR
	μ	min	max	σ		
woQE	0.3988	0.125	1.0	0.2734	48	68
QE$_c$	0.3988	0.125	1.0	0.2734	48	68
QE$_t$	0.4356	0.125	1.0	0.2786	51	71
QE$_w$	0.4822	0.125	1.0	0.1889	65	80
QE$_{wct}$	**0.5189**	0.125	1.0	0.2027	**68**	**83**

Legend: woQE = without query expansion, QE$_w$ = WordNet query expansion, QE$_c$ = code thesaurus query expansion, QE$_t$ = type thesaurus query expansion, QE$_{wct}$= query expansion using the three thesauri; #SQ - number of successful queries; #RFR - number of relevant functions retrieved.



Legend: woQE - without query expansion; QE$_c$ - code thesaurus query expansion; QE$_t$ - type thesaurus query expansion; QE$_w$ - WordNet query expansion; QE$_{\text{wct}}$ query expansion combining all three thesauri (WordNet, code, and type); #RFR - number of relevant functions retrieved.

Figure 2: Number of relevant functions retrieved by each approach in our experiment.

according to **RQ1**. The average recall when this approach was used was 0.4822, that is, around 20% higher than when no query expansion was applied. To check whether the observed difference in terms of this metric was significant, we ran the Wilcoxon test, which indicated that the difference was significant at 90% confidence level (df = 67, p-value = 0.0147). Such results favor the *alternative* hypothesis (H_1-A) that query expansion based on WordNet can improve recall.

We have also checked whether precision was hurt after applying the WordNet query expansion. The difference between the mean precisions was only of 0.005, which was confirmed to be non-significant by running a Wilcoxon test (df = 67, p-value = 0.4938).

Code thesaurus query expansion. By analyzing the results when only the code thesaurus query expansion was used, according to **RQ2**, we reached the following findings: the mean value for recall when this approach was applied was exactly the same as when no expansion is used (0.3988). The Wilcoxon test evidently did not confirm a significant difference at 90% confidence level (df = 67, p-value = 1). Such result favor the *null* hypothesis H_2-*0* that query expansion based on code thesaurus performs similarly with respect to recall. The same also happened with respect to precision.

This is an important result, as WordNet seems to be more effective than the code thesaurus in our scenario. However, this might be explained by the fact that our code thesaurus is still very small (with less than 100 word pairs). Also, considering the functions targeted in our experiment, queries might not have benefited from the code thesaurus. This does not mean that there are no scenarios in which the thesaurus would yield significant results (in fact, we do believe that a code thesaurus might be very important in some situations).

Type thesaurus query expansion. When only the type thesaurus was used, by analyzing our results according to **RQ3**, we also reached interesting findings. The mean value for recall when this approach was used was 0.4356, 9% higher than when no query expansion was applied. The Wilcoxon test showed a statistically significant difference at 90% confidence level (df = 67, p-value = 0.08678). The type expansion approach we adopted is very simple: we only expand numerical and Java Collection types. Thus, we believe that a more sophisticated type expansion approach would perform even better. The observed results favor the *alternative* H_3-A hypothesis, that query expansion based on a type thesaurus can improve recall. Precision was again not hurt when the expansion approach was applied – the difference was of only 0.015 –, which was confirmed by the Wilcoxon test (df = 67, p-value = 0.1976).

WordNet, code, and type thesauri query expansion. Finally, we analyze the results for the query expansion combining the three thesauri, according to **RQ4**. The mean value for recall when this approach was used was 0.5189, 30% higher than when no query expansion was applied[12]. A significant difference was confirmed by the Wilcoxon test (df = 67, p-value = 0.003291). Such result favor the *alternative* hypothesis (H_4-A), that query expansion based on WordNet, code, and type thesauri improves recall. With respect to precision, the expansion approach did not hurt it significantly, either. The observed difference was of only 0.02 (df = 67, p-value = 0.3486).

Since the approach combining the three thesauri and the approach using only WordNet were the ones that attained best results, we performed an additional statistical test to check whether the difference in recall between the two was significantly different. The Wilcoxon test does show a statistically significant difference at 90% confidence level (df = 67, p-value = 0.08678), which confirms that the combined approach is notably better than the others.

5.1 Code Search Based on Simple Name: A Baseline

Other researchers that applied natural language thesauri for code search in the past have not been so successful [11]. We hypothesize that this happened because the expansion approaches were applied mainly to keywords – basically coming from identifiers such as method names –, without considering the interfaces, which can significantly decrease precision and recall (when the number of returned results is limited). Considering this, an interesting baseline to compare to our approach is code search based on simple method name. Therefore, to make such comparison, we have performed searches using the method names suggested by the subjects in our experiment, to check whether our hypothesis confirm, by comparing the use of non-expanded queries, and queries expanded by using WordNet and our code thesaurus. In fact, when we perform the expanded searches using the method name part of the query alone, not only recall is not improved, but in fact it decreases by a small amount (2%). Such difference is not statistically significant, according to the

[12]In fact, we noticed that the code thesaurus did not make any difference in this approach, but we decided to maintain it in the final combination for completion.

Wilcoxon test, which is an evidence that query expansion applied only to keywords – without considering the whole interface – is ineffective in this context.

6. THREATS TO VALIDITY

In this section, we discuss our experiment limitations based on the four categories of threats to validity described by Wohlin et al. [44]. Each category includes several possible threats for the validity of results of an experimental study. For each category, we list possible threats, measures that we took to reduce each risk, and suggestions for improvements in future evaluations.

Internal validity. An internal validity threat that may have affected our experiment was the lack of control of the the subjects' skill. The repeated measures design decreases the probability of this threat affecting our outcomes, since the same user query was used before and after the expansion.

External validity. A characteristic of our experiment that might reduce its external validity is the use of students as subjects. In fact, some studies have shown opposite trends for students and professional developers (*e.g.*, Arisholm and Sjoberg [1]). However, according to other authors, students can play an important role in experimentation in the field of software engineering [4, 17] (specially for an initial evaluation like ours). In any case, 12 professional developers have participated in our experiment (1/3 of our sample), and results were also significant for this group. Therefore, we believe this threat has been circumvented by the inclusion of such group.

Our study included only Brazilian subjects, which might also affect the external validity of our experiment (*i.e.*, our results might generalize to non-native speakers in general). However, even when we consider only the population of Portuguese speaking developers, we have a considerable amount of people that would benefit from our approach. For instance, in Brazil alone there are around 400,000 software developers [13]. Moreover, we believe our results might also be generalizable to native Spanish speakers, because the two languages are very similar in their relation to English[14]. For example, many Spanish-speaking natives would probably pick the word *invert* in the case of the *string reversion* function discussed in Section 3, because in that language the word *invertir* would be the most adequate to be used. Since Spanish is the second most spoken language in the world (coming only after Mandarin)[15], a multitude of other developers could benefit from our approach.

Another threat to the external validity of our experiments is the representativeness of the selected functions. One might argue that the functionalities selected do not represent the population of functions in general. We agree that our results might not scale to more complex functions. However, we believe that auxiliary functions are important to software development in general, as discussed by Lemos et al. [22]. Also, as commented in Section 4, this type of function is among the most popular category of features searched in code search engines [2].

Construct validity. A characteristic of our experiment that could have affected its construct validity is the restriction of the result sets to 100, as discussed in Section 4. Such boundary could have artificially raised precision, as the number of results is narrowed down. To check whether such threat was in fact possible, we have run the experiment without such boundary, to see if the conclusions remain the same. In fact, the relations seem to be maintained when no restrictions are made to the results set. For instance, when comparing the use of non-expanded queries to queries with the combined expansion approach, precision is not significantly changed, and recall is increased, similarly to the results presented in Section 5.

[13] `http://tinyurl.com/ppuaq22` - 09/10/2013.

[14] `http://tinyurl.com/kj5rx58` - 02/05/2014.

[15] `https://www.cia.gov/library/publications/the-world-factbook/geos/xx.html` - 09/11/2013.

7. RELATED WORK

Next, we sample some software reuse and query expansion proposals, comparing them to TDCS and the approach presented in this paper.

Code reuse. An earlier approach similar to TDCS was proposed by Podgurski & Pierce [31]. Behavior Sampling (BS) is a retrieval technique that executes code candidates on a sample of inputs, and compares outputs to an oracle provided by the searcher. However, differently from TDCS, inputs for the desired functions are randomly generated and expected outputs have to be supplied by users. TDCS implements the the ability to retrieve code based on arbitrary tests, an extension to BS considered by Podgurski & Pierce. PARSEWeb [43] is a tool that combines static analysis, text-based searching, and input-output type checking for a more effective search.

Reiss [33] argues that most early reuse techniques did not succeed because they required either too little or too much specification of the desired feature. For instane, full semantic matching requires the specification of too much information, and is thus difficult to accomplish. TDCS uses test cases, which are generally easy to check and provide.

In a recent work, Reiss [33] also incorporates the use of test cases and other types of low-level semantics specifications to source code search. He also implements various transformations to make available code work in the current user's context. However, in his approach, there is no slicing facility and therefore only a single class can be retrieved and reused at a time. Moreover, the presented implementation is a web application, which also requires code results to be copied and pasted into the workspace in an *ad hoc* way. Our AQE approach applied in combination with CodeGenie has the advantage of being tightly integrated with the IDE: code candidates can be seamlessly woven to and unwoven from the workspace. With respect to the transformations presented by Reiss [33], none of them target natural language terms as our AQE approach does. Thus, we believe our approach could also be applied to improve Reiss' implementation.

Hummel et al. [15, 16], were one of the first to explore a TDCS approach. They developed a tool similar to CodeGenie, named Code Conjurer. The tool is more proactive than CodeGenie, in the sense that code candidates are recommended to the user without the need for interaction. However, the slicing facility presented by CodeGenie is more fine grained – at method level –, and Code Conjurer does not implement expansions to with the vocabulary mismatch problem targeted by the AQE approach presented in this paper. In fact, the tool implements an automated adaptation engine, but it deals only with the interface specification (*i.e.*, not with keywords or natural language terms). Thus, our AQE approach could also be used to improve their tool.

Lemos et al. [21] presented an approach that uses tag clouds to suggest keywords to the user before defining the test cases used by TDCS. The difference from the approach proposed here is that the former only deals with name terms (*e.g.*, not with types). Moreover, it also requires user interaction and is based solely on WordNet. The approach presented

in this paper generates expanded queries automatically and also applies type and code thesauri.

Modern CASE tools are bringing sophisticated search capabilities into the IDE, extending traditionally limited browsing and searching capabilities [13, 25, 32, 34, 37]. These tools vary in terms of the provided features, but some common ideas that are prevalent among them are the use of the developer's current context to generate queries and the integration of ranking techniques for the search results. CodeGenie also shares such features, bringing the use of test cases to provide a more sophisticated solution.

Query expansion. The application of Natural-Language Processing (NLP) to code and concept search has been proposed by several earlier approaches. For instance, Shepherd et al. [35] combine NLP and structural program analysis to locate and understand concerns in a software system. The basic difference from our approach is that the concern location is targeted towards code in a local project, not in a large open source code repository. Moreover, Shepherd et al.'s approach [35] requires more user interaction, since similar terms have to be chosen by the user iteratively to construct the expanded queries. Our query expansion approach is automatic and thus does not require user intervention.

Gay et al. [9] have proposed an IR-based concept location approach that incorporates relevance feedback from the user to reformulate queries. The proposed approach, however, is not directed to code reuse, but for concept location within a given project. Moreover, such as in Shepherd et al.'s approach, it requires more user interaction.

More recently, Yang and Tan [46] proposed an approach that identifies word relations in code, by leveraging the context of words in comments and code. The idea is to find words that are related in code, but not in English. For such pairs, NL lexicals cannot help. The code thesaurus used in our AQE approach is in great part formed by word pairs generated with their approach. Therefore, our approach is different in the sense that we use their approach to improve ours, while theirs is restricted to the generation of the code-related thesauri. Howard et al. [14] also present an approach to augment natural language thesauri with code-related terms, and Gupta et al. [10], on the other hand, propose part-of-speech tagging of program identifiers. Both techniques could be used to improve ours.

Haiduc et al. [11] have recently proposed an approach for query reformulation based on machine learning to improve text-retrieval (TR) techniques. The difference from our approach is that their technique is more general, and suited to the context of TR-based concept location in source code. It does not incorporate WordNet and does not handle types, as our approach does (because our context is interface-driven code search). Sisman and Kak [39] also proposed an automatic query reformulation approach for code search, but more specific to the context of bug localization (*i.e.*, their approach is suited for locating files, rather than specific functions, and do not deal with types either, for instance).

Most of the query reformulation approaches that were proposed in the past focus on concept location within a project, and mainly to identify code that must be dealt with in a particular software engineering task. The difference between such approaches and ours is that our goal is to help finding code – more specifically, functions implemented in a set of methods with a given API – inside large open source repositories, with the intent of reusing it. Moreover, several approaches require user intervention to be applied – *i.e.*, are not fully automatic – or do not deal with the specificities of interface-driven code search (*e.g.*, do not handle types).

8. CONCLUSION

In this paper we have presented an automatic query expansion approach for interface-driven code search. Our approach combines the use of three thesauri: WordNet, a coderelated terms thesaurus, and a type thesaurus. Initial evaluation in a controlled experiment showed that our approach can significantly improve the chances of finding relevant functions (by 41%), and recall (by 30%, on average), without hurting precision. Moreover, in our experiment, 20 out of the 36 subjects (55%) attained better results when the composed query expansion approach was applied to their queries (for the rest, results remained the same after applying the expansion); 7 out of the 36 subjects (19%) only found relevant functions when a query expansion approach was applied; and 2 out of the 13 functions (15%) were only found when one of the query expansion approaches was applied. Future work includes enlarging our code thesaurus and incorporating other NLP techniques to improve our approach.

9. ACKNOWLEDGEMENTS

We thank FAPESP for financial support (grants 2010/15540 -5, 2013/25356-2, and 2012/21503-8) and Hill, Pollock, Yang, and Tan for providing word pairs for our code thesaurus.

10. REFERENCES

[1] E. Arisholm and D. I. K. Sjøberg. A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Technical Report 6, Simula Research Laboratory, June 2003.

[2] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *Empirical Softw. Engg.*, 17(4-5):424–466, Aug. 2012.

[3] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. of the FSE 2010*, pages 157–166, 2010.

[4] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25:456–473, 1999.

[5] L. Briand, P. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. pages 412–421. ACM, 1997.

[6] R. Burrows, F. C. Ferrari, O. A. L. Lemos, A. Garcia, and F. Taiani. The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study. In *Proc. of the ISSRE 2010*, pages 329–338, Washington, DC, USA, 2010. IEEE Computer Society.

[7] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, Jan. 2012.

[8] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proc. of the ICSE 2012*, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[9] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Proc. of the ICSM 2009*, pages 351–360. IEEE, 2009.

[10] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proc. of the ICPC 2013*, pages 3–12, May 2013.

[11] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proc. of the ICSE 2013*, pages 842–851, Piscataway, NJ, USA, 2013. IEEE Press.

[12] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. of the UIST '07*, pages 13–22, New York, NY, USA, 2007. ACM.

[13] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. of the ICSE 2005*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[14] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proc. of the MSR 2013*, pages 377–386, Piscataway, NJ, USA, 2013. IEEE Press.

[15] O. Hummel and W. Janjic. Test-driven reuse: Key to improving precision of search engines for software reuse. In S. E. Sim and R. E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 227–250. Springer New York, 2013.

[16] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25:45–52, September 2008.

[17] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28:721–734, 2002.

[18] O. Laitenberger and J.-M. DeBaud. Perspective-based reading of code documents at Robert Bosch GmbH. *Information and Software Technology*, 39(11):781–791, 1997.

[19] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53:294–306, April 2011.

[20] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher. Codegenie: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN OOPSLA*, pages 917–918, New York, NY, USA, 2007. ACM.

[21] O. A. L. Lemos, A. C. de Paula, G. Konishi, J. Ossher, S. Bajracharya, and C. Lopes. Using thesaurus-based tag clouds to improve test-driven code search. In *Proc. of the SBCARS 2013*, 2013.

[22] O. A. L. Lemos, F. C. Ferrari, F. F. Silveira, and A. Garcia. Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming. In *Proc. of the ICSE 2012*, pages 529–539, Piscataway, NJ, USA, 2012. IEEE Press.

[23] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio. Development with off-the-shelf components: 10 facts. *IEEE Softw.*, 26(2):80–87, Mar. 2009.

[24] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, 2009. 10.1007/s10618-008-0118-x.

[25] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proc. of the PLDI 2005*, pages 48–61, New York, NY, USA, 2005. ACM Press.

[26] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[27] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[28] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proc. of the ICSE 2004*, pages 282–292. IEEE, 2004.

[29] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[30] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proc. of the ICSE 2000*, pages 734–737, New York, NY, USA, 2000. ACM.

[31] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303, 1993.

[32] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS - an eclipse plug-in for source code exploration. In *Proc. of the ICPC 2006*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

[33] S. P. Reiss. Semantics-based code search. In *Proc. of the ICSE 2009*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[34] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *Proc. of the OOPSLA 2006*, pages 413–430, New York, NY, USA, 2006. ACM Press.

[35] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. of the AOSD 2007*, pages 212–224, New York, NY, USA, 2007. ACM.

[36] F. Shull, J. Singer, and D. I. Sjøberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[37] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *Proc. of the ICSE 2006*, pages 905–908. ACM, 2006.

[38] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proc. of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 21–. IBM Press, 1997.

[39] B. Sisman and A. C. Kak. Assisting code search with automatic query reformulation for bug localization. In *Proc. of the MSR 2013*, pages 309–318, Piscataway, NJ, USA, 2013. IEEE Press.

[40] M. Sojer and J. Henkel. License risks from ad hoc reuse of code from the internet. *Commun. ACM*, 54(12):74–81, Dec. 2011.

[41] D. Spinellis and C. Szyperski. Guest editors' introduction: How is open source affecting software development? *IEEE Softw.*, 21(1):28–33, Jan. 2004.

[42] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proc. of the ICPC 2008*, pages 123–132, Washington, DC, USA, 2008. IEEE Computer Society.

[43] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proc. of the ASE 2007*, pages 204–213, New York, NY, USA, 2007. ACM.

[44] C. Wohlin et al. *Experimentation in Software Engineering: an Introduction*. Kluwer, 2000.

[45] J. Xu and W. B. Croft. Query expansion using local and global document analysis. In *Proc. of the SIGIR 1996*, pages 4–11, New York, NY, USA, 1996. ACM.

[46] J. Yang and L. Tan. Inferring semantically related words from software context. In *Prof. of the MSR 2012*, pages 161–170, Zurich, 2012. IEEE.

[47] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie. A comparison between software design and code metrics for the prediction of software fault content. *Inf. and Soft. Technology*, 40(14):801–809, 1998.