Can the Use of Types and Query Expansion Help Improve Large-Scale Code Search?

Otávio Augusto Lazzarini Lemos*, Adriano Carvalho de Paula*, Hitesh Sajnani†, and Cristina V. Lopes†
*Science and Technology Department – Federal University of São Paulo at S. J. dos Campos – Brazil
{otavio.lemos, adriano.carvalho}@unifesp.br
‡Donald Bren School of Information and Computer Sciences – University of California, Irvine – USA
lopes@ics.uci.edu

***Abstract*—With the open source code movement, code search with the intent of reuse has become increasingly popular. So much so that researchers have been calling it the new facet of software reuse. Although code search differs from general-purpose document search in essential ways, most tools still rely mainly on keywords matched against source code text. Recently, researchers have proposed more sophisticated ways to perform code search, such as including interface definitions in the queries (*e.g.*, return and parameter types of the desired function, along with keywords; called here Interface-Driven Code Search – IDCS). However, to the best of our knowledge, there are few empirical studies that compare traditional keyword-based code search (KBCS) with more advanced approaches such as IDCS. In this paper we describe an experiment that compares the effectiveness of KBCS with IDCS in the task of large-scale code search of auxiliary functions implemented in Java. We also measure the impact of query expansion based on types and WordNet on both approaches. Our experiment involved 36 subjects that produced real-world queries for 16 different auxiliary functions and a repository with more than 2,000,000 Java methods. Results show that the use of types can improve recall and the number of relevant functions returned (#RFR) when combined with query expansion (∼30% improvement in recall, and ∼43% improvement in #RFR). However, a more detailed analysis suggests that in some situations it is best to use keywords only, in particular when these are sufficient to semantically define the desired function.**

***Index Terms*—software reuse, code search, query expansion**

I. Introduction

A form of software reuse that has recently become popular among developers is querying over, copying, and pasting open source code available on the Web. Some researchers have called it the new facet of software reuse [1–3]. Following current information retrieval systems, most code search tools that support this form of reuse rely mainly on keywords. Users enter terms via an interface that consists of a single input box [4], and the tool returns the most relevant code fragments that match such keywords in their bodies or identifiers (keyword-based code search – KBCS – henceforth). Popular code search engines, such as Open Hub[1], krugle[2], and searchcode[3], follow such a model.

Code search differs from general-purpose document search in many ways. Studies have shown, for instance, that when developers perform code search, they are more interested in finding definitions of functions and their uses rather than code fragments [5–7]. Moreover, when compared to natural language text, source code is scarce in words that describe the entities defined and used in the code [8]. Although these variations are well-known, most code search tools still rely mainly on the keyword-based model of search implemented in general-purpose search engines.

Concerned with this, researchers have proposed more sophisticated ways to perform code search. In particular, some tools support the use of syntactic information of the desired function in the queries, to perform more precise searches. For instance, Sourcerer [9], an infrastructure for large-scale indexing and analysis of open source code, supports searching using interface definitions. A front-end that uses Sourcerer allows users to specify, along with keywords, the return and parameter types of the desired function (called here Interface-Driven Code Search – IDCS). Signature matching [10], proposed long before, is a form of IDCS for searching for functions within a software library, and more recent academic approaches also make use of types to improve the performance of code search (*e.g.*, Strathcona [11], PARSEWeb [12], and Prospector [13])[4]. However, current code search tools available for general use, such as Open Hub, searchcode, and krugle, do not support the use of types in queries.

Although sophisticated approaches to code search could indeed improve its effectiveness, to the best of our knowledge there are few empirical studies that evaluate them in comparison with the traditional model of search engines. Such evaluation is important, because it could be the case that adding more syntactic or semantic information to the search could further hinder its performance rather than improving it. For instance, when information such as return and parameter types of the desired function are used in the queries, the search space might narrow down too much, making searches return few or no results at all. In fact, in a recent code search experiment with a test repository of 100 Java projects and more than 2,000,000 methods – the same repository used in this paper –, we have found that the search space is reduced by more than 90% when the return type of the desired function is set to *int*, for example [18]. This happens because less than 10% of the methods in the repository return that type.

Another problem that affects information retrieval in general and code search in specific, is the vocabulary mismatch problem (VMP). VMP states that the likelihood of two people choosing the same keyword for a familiar concept is only between 10-15% [19]. Since code search is carried out using keywords as well, VMP is also relevant in this context. A straightforward and effective way to circumvent such problem

[1]https://code.openhub.net/ - 03/09/2015.
[2]http://www.krugle.com/ - 03/09/2015.
[3]https://searchcode.com/ - 03/09/2015.

[4]Other researchers have also proposed the use of semantic information in the searches, such as test cases [14–16], contracts [15], or even lightweight specifications [17].

is to automatically expand queries with terms similar to the ones originally selected by the users (*i.e.*, automatic query expansion – AQE [4]). So, another important question in this context is to compare the effectiveness of traditional keyword-based code search with more specific code search approaches (such as IDCS) in the presence of AQE approaches.
Goal: In this paper we describe an experiment we conducted comparing KBCS with IDCS in the context of large-scale code search of functions implemented in Java. The idea is to see how KBCS compares to IDCS when searching for specific functions implemented in methods; that is, whether using specific syntactic information of the desired function – in our case, return and parameter types – could help or hinder the search performance, when compared to searching using only keywords matched against the functions' identifiers. We also compare KBCS with IDCS when an AQE approach with a natural language thesaurus (*i.e.*, WordNet) and a simple type thesaurus is applied. Such comparison is important, as some researchers have argued that natural language thesauri are generally not appropriate for software engineering search approaches in general [20–23].
Findings: Our study provides evidence that IDCS can significantly improve code search when compared to KBCS. In particular, IDCS in its best configuration improved recall by ~30%, when compared to KBCS. Moreover, query expansion was much more effective with IDCS: we observed an improvement of ~23% in recall for this approach. On the other hand, for KBCS there was no observable improvement in our experiment. This might explain why some researchers have found query expansion with natural language thesauri to be ineffective in software engineering search approaches: because it was used within a keyword-only model of search.

The remainder of this paper is structured as follows. Section II provides background information about the topics targeted in this paper; Section III outlines the experimental setting of our study; and Section IV presents the results of our experiment and its in-depth analysis. Section V discusses limitations of our study; and Section VI summarizes published research related to ours. Finally, Section VII presents the main conclusions of our experiment.

II. 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 [24]. In this endeavor developers often resort to general purpose search engines like Google, providing keywords to match the implementations they desire. As described earlier, specialized code search engines like Open Hub, and few others have improved the chances of finding relevant code by indexing source code only, and providing more advanced filtering (*e.g.*, by programming language and search within method names only). However, these search engines are still very much tied to keywords; for instance, users cannot specify the return or parameter types of the desired function in the query.

A. Interface-Driven Code Search

Recently, researchers have proposed more advanced infrastructure to analyze and index source code, and thus support more sophisticated code search. Sourcerer is one of these efforts [9]. In summary, given a set of Java projects with available source code, Sourcerer can do the following: (1) automatically collect and store the code in a repository with a standardized directory structure; (2) find and perform an in-depth structural analysis of the available source code (*e.g.*, dependence analyses); and (3) store the code structure information in a centralized database. With this infrastructure it is possible to perform several types of mining tasks within the target repository, including code search.

Sourcerer stores essential structure information of the target source code in its database. For example, the database can tell which packages contain which classes, which classes contain which fields or methods, which methods are called by which methods, and what are the return and parameter types of a given method. Therefore, by using such infrastructure it is possible to perform searches in which information such as the return and parameter types of the desired function are specified in the queries. For instance, a user interested in the implementation of a function to *merge* a list of strings into a single string can search for a method that, given a *list of strings* (in Java: `List<String>`) returns the merged `String` and contains the term *merge* in its identifier.

In order to perform this implementation of Interface-Driven Code Search, Sourcerer receives Solr-formatted[5] queries. Figure 1 presents one such query for the *string merging* function mentioned before. We also highlight the query fields that are included in the KBCS and IDCS implementations used in the experiment described in this paper[6]. It is important to note that in these implementations the keywords are matched against function's identifiers (*i.e.*, method names), and not their bodies. This is done because in our context we are looking for specific functions, so it is reasonable to expect that the functions' names would contain the keywords given by the users[7]. This type of filtering is also supported by available code search engines such as Open Hub. The `mdef:<search terms>` filter can be used to produce such an effect in that tool.

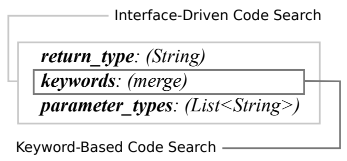
Interface-Driven Code Search
***return_type**: (String)*
***keywords**: (merge)*
***parameter_types**: (List<String>)*
Keyword-Based Code Search

Fig. 1. Solr query example for a *string merging* function when KBCS and IDCS is used.

[5]http://lucene.apache.org/solr/ - 03/16/2015.

[6]In this implementation of IDCS, we only consider return type, method name, and parameter types to form an interface. Other elements could also be handled, such as field types of the owning class and thrown exceptions. However, we believe the information used here is adequate for the proposes at hand. By using other elements of the interface, the search space might become too narrow, resulting in poor recall [18]. Also note that partial matching is performed for types. For example, a query that contained `(int)` in its parameters' field would also match a method whose interface contained parameters `(int, double, double)`. Moreover, the order in which the types appear is irrelevant: the parameter types list only have to contain all types that are included in the query. Also, if parameter types are found among the keywords' list they are removed to avoid redundancy.

[7]Sourcerer also supports matching against full qualified names. However, since such feature is not available in available code search engines, we decided to use the method name only, to be able to perform an evaluation closer to the current practical use.

It is important to note that approaches similar to IDCS can be found in the literature as early as 1995 (with *signature matching* [10]). Signature matching, for instance, is a form of IDCS for searching for functions within a software library, in which the user can specify input and output types of the desired function in the query. More recently, other tools such as Prospector [13], PARSEWeb [12], and Strathcona [11], have also included types in queries to improve the performance of code search (a more thorough summary of code search approaches will be discussed in Section VI). IDCS is also used internally with Test-Driven Code Search [14, 18] (and similar approaches [15, 16]), so that functions that are matched can be easily adapted to the local context before running test cases.

However, to the best of our knowledge, there are no studies that empirically investigate the performance of these approaches that use types against the traditional keyword-based search model, also addressing query expansion. In this paper we describe one such experiment involving IDCS within a contemporary context of large-scale code search of functions implemented in Java.

B. Query Refinement

The relative ineffectiveness of IR systems is mainly caused by the inaccuracy with which a query formed by few keywords can model the actual user information need [4]. Another known issue is that, in most collections, the same concept may be referred to with different words. This issue, known as *synonymy* (or vocabulary mismatch problem), impacts the recall of most IR systems [25]. This is also true in the context of code search, because keywords are still required.

Query refinement (QR) methods have been proposed to deal with this problem. In general, QR methods are classified as *global* or *local*. Global methods expand query terms independently of the results returned from the original query, so that changes in its wording will cause the new query to match other terms with similar semantics. Local methods, on the other hand, adjust a query relative to the documents that originally appear to match the initial query [25].

Automatic query expansion (AQE) is a well known global method to overcome synonymy [4]. It augments the user's original query with new features with similar semantics. The approach used in the experiment described in this paper [18] is a global approach that does not require running the query prior to expanding it: the query is first expanded with similar terms, and then executed to find relevant functions. This strategy generally improves performance, as queries are run only once. Another advantage is not requiring additional user input [25].

The AQE approach used in our experiment can be configured with different thesauri [18]. In this paper, we configured it to be used with WordNet [26], a lexical database for the English language, and a simple type thesaurus that expands numerical and Java collection types. In the above example where the user is looking for a function to merge a list of strings into a single string, the expanded IDCS query with the two selected thesauri would look like the following:

return_type: *(String)*
keywords: *(merge^10* $\vee$ *unite* $\vee$ *...)* $\wedge$
!(divide $\vee$ *disunify)*
parameter_types: *(List<String>* $\vee$ *List* $\vee$ *...)*

Note that the query field where keywords are used is expanded with synonyms. More weight is given to the original terms selected by the user with the Lucene[8] boost factor, so that candidates that possess such terms are given a higher rank (in the example, *merge^10*). This is an evolution of the original algorithm presented previously [18]. When there are multiple keywords, a conjunction is used among them (*e.g.*, see Table VII, query #3), and a disjunction is used among the synonyms (*e.g.*, *merge* OR *unite*). In this way, a candidate is matched when it contains at least one term related to each of the concepts used in the query. The type thesaurus allows for a relaxed matching of the types. In the example, `List<String>` is expanded with other types of lists contained within the Java API (including the alternative where the type parameter `<String>` is removed). In queries, antonyms are also used to filter out unwanted results that might implement the opposite of the desired function (*e.g.*, *unzip* when *zip* is being searched).

III. Study Setup

Our goal is to investigate the difference in effectiveness of IDCS when compared with KBCS. The main question is whether including the return and parameter types of the desired function in the code search queries can improve or hinder its effectiveness. We also want to compare the performance of KBCS with IDCS when an AQE approach with WordNet and a type thesaurus is used. This is important, as some researchers have argued that natural language thesauri are generally not appropriate for code search and software engineering in general [20–22]. Also, as a secondary evaluation we check whether the AQE approaches can improve recall for KBCS and IDCS individually. Thus, in our study, we are interested in the following research questions:

In the context of large-scale code search of methods implemented in Java, with respect to the comparison of KBCS with IDCS:

RQ1. Does IDCS improve recall when compared to KBCS?
RQ2. Does IDCS improve recall when compared to KBCS, when AQE based on natural language and type thesauri is used?

In the context of large-scale code search of methods implemented in Java, with respect to query expansion:

RQ3. Does AQE based on natural language thesaurus improve recall for KBCS?
RQ4. Does AQE based on natural language and type thesauri improve recall for IDCS?

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

TABLE I
Hypotheses formulated for our experiment.

	Null hypothesis (0)	Alternative Hypothesis (A)
H_1	$R_{IDCS} = R_{KBCS}$	$R_{IDCS} > R_{KBCS}$
H_2	$R_{IDCS_{AQE_{wt}}} = R_{KBCS_{AQE_w}}$	$R_{IDCS_{AQE_{wt}}} > R_{KBCS_{AQE_w}}$
H_3	$R_{KBCS_{AQE_w}} = R_{KBCS}$	$R_{KBCS_{AQE_w}} > R_{KBCS}$
H_4	$R_{IDCS_{AQE_{wt}}} = R_{IDCS}$	$R_{IDCS_{AQE_{wt}}} > R_{IDCS}$

Legend: H = Hypothesis, R = recall, IDCS = Interface-Driven Code Search; KBCS = Keyword-Basd Code Search; AQE_w = Automatic query expansion with WordNet; AQE_{wt} = Automatic query expansion with WordNet and a type thesaurus.

[8]http://lucene.apache.org - 03/16/2015.

A. Repository, Functions, and Queries

Repository. Fraser and Arcuri [27] argue that several authors that propose novel test data generation techniques do not conduct sound empirical studies to provide evidence for the usefulness of their approaches. Case studies on such topic tend to be either small or biased toward a specific kind of software [27]. To cope with this problem, these researchers have randomly selected 100 Java projects from SourceForge[9] in what was probably a first attempt to run a statistically sound experiment in test data generation. According to the authors, the resulting benchmark, called SF100, is statistically sound and representative of open source projects[10]. 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 can introduce bias to the study. Concerned with this problem, and to construct a manageable and unbiased code base, for our experiment we used Sourcerer to index Fraser and Arcuri's repository. SF100 thus forms the target repository for the experiment described in this paper.

Functions. For our experiment, we have selected a set of functions to be searched in the repository. The idea was to simulate users looking for implementations of specific functions. To avoid bias, we chose functions that were also used in previous code search studies [14, 15, 18, 28].

To have a list of realistic features from that first set, we selected only those that appeared at least once in our target repository. In this way we could be sure 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 matches. This procedure was important to be able to measure query successfulness and recall.

The features are *auxiliary functions*, that is, supportive actions of software systems that comprise 10-200 lines of code [29]. According to a recent study that explored a large log of queries submitted to a code search engine, this type of function – called there *programming tasks* – is the one most frequently targeted by developers [30]. Another study showed the importance of auxiliary functions to software development in general [29]. Therefore, we believe these features are adequate for the purposes at hand; that is, they represent important implementations that are in fact searched by developers. The functions selected for our study together with the frequency in which they appear in the target repository are listed in Table II.

Queries. To adequately evaluate our approach, we had to look into queries that users would really generate while performing keyword-based and interface-driven code search; that is, we could not guess which keywords and interfaces they would pick without biasing our study. Therefore, we have conducted a survey to collect realistic interface definitions and keywords that users would use in their queries. Subjects were presented with the functions' descriptions, and were asked to define what interfaces and keywords they would use to search for those functions. Since we are interested in KBCS and IDCS, we asked them to guess the return type, name, and parameters types of the entry point method of the function that would implement the desired behavior. Keywords were extracted from the method name, for both approaches. Then, to measure the performance of those queries, we conducted the searches by using the types and keywords chosen by the subjects. 36 subjects responded to our survey; from them, 24 were senior Computer Science undergraduate students and 12 were professional developers. Each of them created queries for all target functions.

[9]http://sourceforge.net/ - 04/22/2015.

[10]http://www.evosuite.org/sf100/ - 04/22/2015.

TABLE II
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
14	Converting camel case strings to phrases	1
15	Blurring an image	1
16	Capturing the screen into an image	1

B. Metrics

To evaluate the performance of the queries when KBCS or IDCS is used, we adopted recall (R), a traditional measure used in the evaluation of information retrieval systems. Precision was not included because we are only considering the top 10 results returned for each query. Formally, consider q a query collected in the experiment and r the number of top candidates that were considered when running the query. In our context, such a metric is defined as follows:

$$\mathbf{R_r(q)} = \frac{|\text{retrieved functions} \cap \text{relevant functions}|}{|\text{relevant functions}|}$$

We also measure *query successfulness*, which computes whether a given query succeeded or failed in the task of matching at least a single relevant function. The *query successfulness* S of a given query q, when the top r results are considered, is defined as follows:

$$\mathbf{S_r(q)} = \begin{cases} 1 & \text{when at least one relevant function is matched} \\ 0 & \text{otherwise} \end{cases}$$

We believe S is an important metric 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. Moreover, by having two different metrics we hope to have a more complete evaluation of the competing approaches.

Other metrics used in our experiment are the *number of successful queries* (#SQ), which refers to the number of queries that retrieved at least a single relevant function (*i.e.*, $\sum_{i=1}^{n} S_r(q_i)$, where n is the number of queries considered in the experiment); the proportion of queries that are successful only with a given approach p (*i.e.*, $\frac{\#SQ}{n}$); and the *number of relevant functions retrieved* (#RFR; *i.e.*, $\sum_{i=1}^{n} RF_r(q_i)$, where $RF(q)$ is the number of relevant functions retrieved for q).

For the statistical tests we have set r to 10; that is, we consider the top 10 results returned by each query. Other code search experiments also adopted such a threshold (*e.g.*, [5, 31]). We believe this is an adequate number of top results to be considered, since it is difficult to suppose a user would go over more than 10 results in search of a relevant function. Also, for all measurements we only considered the queries that matched at least a single relevant result by using one of the approaches. This is quite reasonable as we are not interested in measuring how good users are in defining queries, but how the approaches compare to each other when a reasonable query is designed. For ranking we rely on Solr algorithms for all approaches.

C. Statistical Analysis, Experimental Design and Procedure

For the conducted experiments, we adopted the *repeated measures* experimental design, where each query is evaluated while first applying KBCS and later adding types and expansions (*i.e.*, IDCS and AQE). Such type of design supports more control over the variability among the subjects (in our case, the queries and users) [32]. Each query was executed while using KBCS and IDCS, and with and without AQE. In this case, to test for statistical significance, *paired* hypothesis tests can be used to compare measures within items rather than across them. Because of this, paired tests are considered to improve precision when compared to unpaired tests [32].

For *recall*, which is a continuous metric, we applied the Wilcoxon/Mann-Whitney non-parametric signed-rank paired test to check the significance of the observed differences (after a Shapiro-Wilk test indicated that the observations in our experiments did not follow a normal distribution). Since query successfulness is binary, for such metric we adopted the McNemar's χ^2 test to determine whether the reached proportions of successful queries for each approach were significantly different.

In our study, we adopt the standard confidence level of 95%. Our analyses thus consider p-values below 0.05 significant. For statistical tests we used the R language and environment[11].

IV. Results and Analysis

Table III shows the main results of our experiment[12]. In particular, we show the measured average recall[13], total number of relevant functions retrieved (#RFR), proportion of queries that were successful only with the given approach (p), and the total number of successful queries (#SQ) when each approach is applied. To provide a general view of the outcomes, the graph presented in Figure 2 summarizes our results in terms of the total #RFR when each approach was used.

[11]http://www.r-project.org/ - 23/03/2015.

[12]All experimental data can be found at http://tinyurl.com/o72ofeo.

[13]Recall outcomes are relatively low compared to other IR systems. This happens because, in our case, it is almost impossible to return all – or most – relevant functions for a given query, because queries are very specific toward an implementation of the function. For instance, a user that speculates that an *image resize* function would contain an `Image` parameter will never find the alternative implementations that manipulate an `Image` field rather than parameter.

TABLE III
Summary of the results of our experiment.

Expansion	Approach	Avg. Recall	#RFR	p	#SQ
None	KBCS	0.3008	65	30.35%	62
	IDCS	0.3025	70	27.67%	59
WordNet	KBCS	0.3008	65	30.35%	62
WordNet + type	IDCS	0.3939	93	44.64%	80

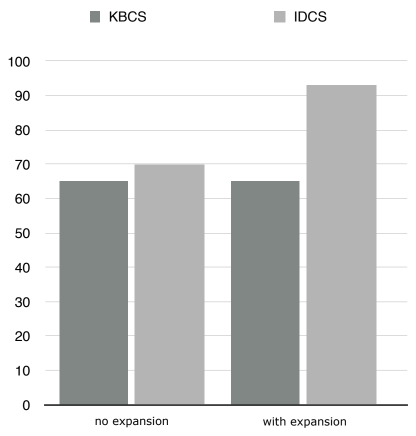

Fig. 2. Total number of relevant functions returned for each approach.

As shown in the table, the average recall and #RFR metrics were improved when IDCS was applied, in all contexts. Although there is very little improvement in recall without any expansion, the queries attained a better recall for IDCS when types and query expansion were used in combination as compared to KBCS (~30% improvement). However, #SQ and p were lower for IDCS when no expansion was used. Also, it seems that query expansion have improved IDCS, but not KBCS. Next, we analyze our results in the light of the research questions established in Section III.

A. KBCS vs. IDCS

First, we analyze our results when KBCS and IDCS are applied with no query expansion, according to **RQ1**. Recall was improved very slightly when the types were added to the initial KBCS queries (~0.5% improvement). To check whether the statistical relevance of the observed difference in terms of this metric, we ran the Wilcoxon test, which did not indicate a significant difference at 95% confidence level (p-value = 0.2328). Such result favors the *null* hypothesis (H_1-*0*) that IDCS performs similarly to KBCS when no query expansion is applied. It is worth noting that although addition of types did not significantly improve the recall, they did not hinder the performance either. It implies that the type information in the query is not noise but might have valuable information that can be harnassed along with other expansion approaches to improve the performance - which is indeed validated from the results of **RQ2**.

Now, with regards to **RQ2**, which refers to the comparison between KBCS and IDCS when a combined query expansion approach with WordNet and a type thesaurus is used for IDCS, we also reached interesting results. This time the recall of queries was further improved, when compared to KBCS (~30% improvement). The Wilcoxon test this time indicated a significant difference at 95% confidence level (p-value = 0.00484). Such result favors the alternative hypothesis (H_2-A that IDCS outperforms KBCS when WordNet and type thesaurus query expansion is used. This outcome is important as it indicates that a more relaxed type matching can further improve the performance of IDCS.

B. Query Expansion Applied to KBCS and IDCS

Now with respect to our secondary evaluation pertaining query expansion applied to both KBCS and IDCS, our results were also compelling. With respect to **RQ3**, which refers to the evaluation of AQE with WordNet when applied to KBCS, it is clear from the results in Table III that in terms of recall AQE did not affect the performance of the searches, it remained the same after applying AQE (avg. recall = 0.3008). The statistical test confirms a non-significant difference at 95% confidence level (p-value = 1).

When AQE was applied to IDCS, it influenced the recall of queries. With regards to **RQ4**, when AQE with WordNet and the type thesaurus was applied, recall was improved by around 30%, on average. The Wilcoxon test shows a significant difference at 95% confidence level (p-value = 0.00001709).

Such results are very important, because they show that AQE with a natural language thesaurus does not seem to be effective along with a keywords-only search approach. However, when types are used, since the search space is first narrowed down to functions with a specific interface, the AQE approach is effective in indicating alternative terms to the queries. This might be an explanation why some researchers have argued that natural language thesaurus seem to be ineffective for query expansion in software engineering [20–23]: because they applied it within a keyword-based search model. Our results suggest that more filtering is required in order to benefit from AQE with natural language thesaurus in this context. Outcomes also indicate that it is effective to perform a more relaxed matching with respect to the types, when they are applied within IDCS (results for when only WordNet is used for IDCS are not shown here, but they are worse than when both thesauri are used, but slightly better than IDCS with no expansion). This is intuitive as restricting the search only to specific types might filter out relevant functions that implement the desired functionality but with slightly different types.

C. Query Successfulness Analysis

Our initial results show that, in order to attain better recall in code search, in general it is better to use types in the queries than not to use them, in particular while applying AQE. However, a question that should be asked is the following: should types *always* be included in the queries? Or are there situations where only keywords should be used and types avoided? In this section we perform a more in-depth analysis using the *query successfulness* metric to pursue such questions.

Tables IV and V present contingency tables for the number of successful queries when each approach is applied: without using query expansion and with WordNet and type thesaurus query expansion. These tables show the number of queries that, for instance, succeeds with one approach and fails with the other.

By looking at Table IV we can see that, in terms of query successfulness, IDCS and KBCS performed very similarly.

TABLE IV
CONTINGENCY TABLE FOR THE NUMBER OF SUCCESSFUL QUERIES WHEN KBCS AND IDCS ARE USED WITH NO QUERY EXPANSION.

		IDCS		
		success	**failure**	**total**
KBCS	**success**	28	***34***	62
	failure	***31***	19	50
	total	59	53	112

TABLE V
CONTINGENCY TABLE FOR THE NUMBER OF SUCCESSFUL QUERIES WHEN KBCS AND IDCS ARE USED WITH QUERY EXPANSION USING WORDNET AND A TYPE THESAURUS.

		IDCS – WordNet+type		
		success	**failure**	**total**
KBCS – WordNet	**success**	30	***32***	62
	failure	***50***	0	50
	total	80	32	112

In particular, when no expansion was used, 34 queries were successful with KBCS and failed with IDCS (that is, found relevant functions only when KBCS was applied); and 31 queries were successful with IDCS and failed with KBCS. Differences are not so high in the presence of query expansion either: 32 with KBCS vs. 50 with IDCS. McNemar's χ^2 tests indicate that all differences are not statistically significant at 95% confidence level (p-values: 0.8041 and 0.0605). Such results indicate that there can be situations in which it is better to avoid the use of types in the queries.

Analysing further, we compare the number of successful queries of IDCS with no expansion with IDCS when using WordNet and type thesaurus query expansion. The results are shown in Table VI. Note that we found there are no cases in which it is better not to use expansion (0 queries were successful only with IDCS with no expansion and not with the version where expansion is used). The McNemar's χ^2 test in this case indicates a very significant difference, at as high as 99% confidence value (p-value = 0.0001).

TABLE VI
CONTINGENCY TABLE FOR THE NUMBER OF SUCCESSFUL QUERIES COMPARING IDCS WITH NO EXPANSION AND IDCS WITH QUERY EXPANSION USING WORDNET AND THE TYPE THESAURUS.

		IDCS – WordNet+type		
		success	**failure**	**total**
IDCS	**success**	59	***0***	59
	failure	***21***	32	53
	total	80	32	112

We have analyzed the discordant pairs (*i.e.*, cases that were successful with one approach and not the other, and vice-versa) for the three first contingency tables, to try to see example situations in which it is better to use types and situations in which it is worse. Table VII shows five example queries where such discordances happened.

Queries #1 and #2 of the table show situations where IDCS succeeds and KBCS fails in the task of finding at least one relevant function implementation. We believe this happens because the concepts referred to in the keywords themselves are too broad, so undesired methods are matched and relevant implementations might be pushed down in the list of returned

TABLE VII
EXAMPLE QUERIES AND THEIR PERFORMANCE WITH RESPECT TO QUERY SUCCESFULNESS.

#	KBCS (no expansion)		IDCS (no expansion)		KBCS (WordNet expansion)		IDCS (WordNet + type expansion)	
	q	**S_{10}(q)**	**q**	**S_{10}(q)**	**q**	**S_{10}(q)**	**q**	**S_{10}(q)**
1	***keywords***: (rotate)	0	***return_type***: (void) ***keywords***: (rotate) ***parameter_types***: (double)	1	***keywords***: (rotate^10 ∨ revolve ∨ goAround …)	0	***return_type***: (void) ***keywords***: (rotate^10 ∨ revolve ∨ goAround ∨ …) ***parameter_types***: (double ∨ float ∨ long ∨ int ∨ …)	1
2	***keywords***: (reverse)	0	***return_type***: (String) ***keywords***: (reverse) ***parameter_types***: (String)	1	***keywords***: (reverse^10 ∨ invert ∨ reversal ∨ …) ∧ !(obverse ∨ forward)	0	***return_type***: (String) ***keywords***: (reverse^10 ∨ invert ∨ reversal ∨ …) ∧ !(obverse ∨ forward) ***parameter_types***: (String)	1
3	***keywords***: (image) ∧ (resize)	1	***return_type***: (Image) ***keywords***: (resize) ***parameter_types***: (Image)	0	***keywords***: (image^10 ∨ picture ∨ figure ∨ …) ∧ (resize^10)	1	***return_type***: (Image) ***keywords***: (resize^10) ***parameter_types***: (Image)	0
4	***keywords***: (screen) ∧ (capture)	1	***return_type***: (void) ***keywords***: (screen) ∧ (capture)	0	***keywords***: (screen^10 ∨ test ∨ screenOut ∨ …) ∧ (capture^10 ∨ enamour ∨ trance ∨ …)	1	***return_type***: (void) ***keywords***: (screen^10 ∨ test ∨ screenOut ∨ …) ∧ (capture^10 ∨ enamour ∨ trance ∨ …)	0
5	***keywords***: (unite)	0	***return_type***: (String) ***keywords***: (unite) ***parameter_types***: (List<String>)	0	***keywords***: (unite^10 ∨ join ∨ merge ∨ …) ∧ !(divide ∨ disunify)	0	***return_type***: (String) ***keywords***: (unite^10 ∨ join ∨ merge ∨ …) ∧ !(divide ∨ disunify) ***parameter_types***: (List<String> ∨ List ∨ …)	1

Legend: $S_{10}(q)$ = query successfulness of q within the top 10 results (*i.e.*, whether q matched at least a single relevant function within the first 10 returned candidates).

candidates, with KBCS. In query #1, since *rotate* is a term that may refer to the rotation of things other than images, KBCS fails to match a relevant function. For instance, it matches method `void SorterTemplate.rotate(int, int, int)`, which is an auxiliary function used by sort methods to rotate contents of an array, not to rotate images. In the IDCS version of the query, since the user speculated that the desired implementation should have a `double` parameter – probably to define the rotation degree –, the relevant method `void Graphics2D.rotate(double)` is returned within the top 10 results for such a query, thus making it succeed. A similar situation happens when query #2 is run with KBCS: since *reverse* may also refer to the reversing of things other than strings, it ends up matching irrelevant implementations. For instance, it matches method `void TLongArrayList.reverse()`, which reverses items from a list, not characters from a string. When the user defines that the return and parameter types of the function should be `String`, the IDCS query succeeds in matching the method `String StringUtils.reverse(String)`, which does reverse strings (*i.e.*, a relevant implementation). It appears that, for some functions, its full meaning is only attained when their types are considered. Another example from our repository is method `BufferedImage MapPreview.scale(BufferedImage)`. Note that the method name only cannot tell that it refers to an *image scaling* function.

Queries #3 and #4, on the other hand, show situations where KBCS succeeds and IDCS fails. We believe that might happen because, at this time, the keywords themselves are sufficient to describe the desired functions, so types, when wrongly guessed, end up getting in the way of matching relevant implementations[14]. Moreover, it also appears that when the types of the desired function are complex, the performance of IDCS might be hindered. In the case of query #3, by using keywords *resize* and *image*, KBCS matches method `void ImageUtils.resizeImage(String, String)`, which is in fact relevant to the query. However, when the user speculates that the function must contain an `Image` parameter, for IDCS, no relevant functions are matched. It should also be noted that the user programming experience must also impact on the success of IDCS or KBCS: in this case, the subject is a student who did not realize that implementations of the resize image function could manipulate an `Image` field rather than receive it as a parameter (although there is a version in the repository that does receive a `BufferedImage`, but with a different name – *scale* –, as exemplified before).

A similar situation happened with query #4. Since *screen* and *capture* are terms sufficiently semantically rich to define the desired function, KBCS succeeds in matching the relevant method `BufferedImage Robot.createScreenCapture(Rectangle)`. The IDCS version of the query fails to match a relevant function because the user speculated that the method should return `void`, which is not the case for relevant functions in our repository.

We added query #5 to show an example where query expansion is effective in improving the performance of code search. Note that IDCS with no expansion fails in finding a relevant function that implements joining a list of strings, when the keyword *unite* is used, even with the use of types. However, when WordNet expansion is applied in conjunction with types, the relevant method `String StringUtil.merge(List)` is matched, because the original keyword is expanded with the term *merge*, and the types help pinpoint a relevant implementation.

Our analysis shows that the use of types is generally good to improve recall in code search. However, users should also include them in the queries with care, as there are situations in which they might affect the search negatively. Our results seem to suggest that the best strategy to apply is to first start with keywords only and then refine the queries by including types incrementally, whenever relevant implementations fail to

[14]One might argue that it is only a question of adding more keywords to the query. We do not believe this is the case, because there are queries in our experiment in which the user selected a significant number of keywords, and KBCS was not successful either. For instance, in one of the queries, three terms were selected (*decode*, *url*, and *string*), and KBCS was not able to return a relevant implementation.

appear among the top results. In fact, when we simulate such a strategy with our data – by selecting the best approach for each query – recall is raised to ∼0.65. Query expansion seem to be appropriate to use along with types, as such combination seems to impact positively in the performance of code search.

V. Study Limitations

Based on the categories of threats to validity described by Wohlin et al. [33], in this section we discuss our experiment limitations. We list possible threats, measures taken to reduce risks, and improvement suggestions for future evaluations.

Internal validity. In our experiment, there is no control over the subjects' skills (*e.g.*, with Java programming). However, the repeated measures experimental design decreases the risks of this threat affecting our outcomes, because the same user information to form a query was used before – with KBCS – and after – with IDCS.

External validity. The external validity of our experiment could be reduced by the use of students as subjects. Some studies have shown evidence that in some cases students show opposite trends compared to professional developers (*e.g.*, [34]). However, other authors argued that students can play an important role in software engineering experiments [35, 36]. In any case, a measure we took that can effectively reduce the impact of such threat is the participation of 12 professional developers in our experiment (1/3 of our sample). Therefore, we believe the effect of such threat has been mitigated by the inclusion of the group of professionals.

The possible lack of representativeness of the functions selected in our experiment could also affect its external validity. It could be argued that the selected functionalities do not generalize to the population of software functions. We agree that more complex functions might impact the results we attained in our experiment. However, as commented in Section III, auxiliary functions play an important role in software development in general [29]. Also, as discussed in the same section, these type of functions are among the most popular categories of features targeted by users in code search engines [30].

Construct validity. A characteristic of our experiment that could have affected its construct validity would be the use of only recall to measure the performance of code search. To avoid this threat, we added a second metric – *query successfulness* – that allowed us to perform a more precise analysis of the outcomes. The second metric made possible observing that, although types do improve recall in general, in some situations it is better to avoid them in queries.

Another problem that might have affected our conclusions was the procedure adopted to establish a gold set for our repository. Since it was done manually – we had to go through the repository performing manual inspections –, we might have failed to include relevant functions that were there but we could not find, or mistakenly identified irrelevant functions as relevant. The repeated measures design was a way to reduce such a threat. Since the same user information was used in the different query versions (either KBCS or IDCS, for instance), if our gold set has flaws, it affects both approaches so the impact of the issue is alleviated.

VI. Related Work

Code search for reuse. Since the late 1960's, software reuse has been a widely explored topic in software engineering, both from an academic and from a practical perspective [15, 37]. There are many characteristics of reusability that make it a hard problem, including creating reusable code, finding code to reuse, and adapting reusable code to the new application [15]. Several approaches to tackle such aspects have been proposed, but only recent work explore the vast quantity of code available in open source repositories. It is clear that this novel approach to code search and reuse differs significantly from traditional local-scale search, where programmers were more interested in code stored within small repositories, written by local teams of developers, and often targeted towards maintenance tasks, such as debugging and refactoring [31].

In the 1990's, program syntax and semantics were commonly explored as a means to enhance the local search of reusable components [15]. Earlier work explored types as search keys in function libraries, but targeted functional programming languages (*e.g.*, [38], and Hoogle[15]). Zaremski et al. [10] presented an IDCS method that included signature information derived from components in the queries. Such approach, named Signature matching, was later extended with the use of more formal semantics, like λ-prolog and Larch-based specifications [39]. Podgurski and Pierce [40], on the other hand, developed Behavior sampling, a code retrieval technique based on interface specification and input and expected output sample values. Most of these earlier approaches were developed for smaller software libraries based on procedural or functional languages (*i.e.*, local-scale code search). Therefore, results achieved by the evaluation of such approaches are not directly applicable here. Our paper thus updates evidence about the use of types in code search in the context of Java software within large repositories.

More recent code search tools have also included the use of types to improve the performance of code search. PARSEWeb [12], for instance, combines static analysis, text-based searching, and input-output type checking for a more effective search. However, such approach is essentially different from the IDCS approach evaluated in this paper, because its goal is to suggest method-invocation sequences to reach destination types from source types given in a query. Our IDCS approach, on the other hand, targets the reuse of self-contained function implementations. Moreover, PARSEWeb is evaluated against other similar approaches that use types as well (Prospector [13], XSnippet [41], and Strathcona [11]), and not against a keyword-based code search approach, such as the evaluation presented in our paper. Query expansion is seldom explored within these tools.

Test-Driven Code Search (TDCS) is an advanced approach to code search where information available in tests are used to perform searches. Candidates that match such information are woven into the local environment and run against the tests to verify suitability [14][16]. Note that IDCS is performed as a stage of TDCS: the interface of the desired method is extracted from the tests to find the initial set of candidates. The evaluation performed in this paper is thus relevant to TDCS; since our results point out to the effectiveness of using interface information in queries, it is also an evidence for the applicability of TDCS. In a previous work we have also evaluated the use of query expansion within the con-

[15]https://www.haskell.org/hoogle/ - 04/17/2015.

[16]Similar approaches were proposed by other researchers [15, 16, 42]

text of IDCS [18]. However, the aim of that study was to compare query expansion approaches only with IDCS, and not to compare it with a keyword-only code search approach. Moreover, the IDCS approach used in this paper contains a significant evolution from the original algorithm: the boosting of initial terms selected by users for a query, when AQE is applied. Also, in the previous experiment, a threshold of 100 was used for the size of the result set to be considered. In this paper we adopt a more realistic threshold: following other studies [13, 31] we consider only the top 10 candidates returned.

Hummel et al. [43] have conducted a study to evaluate different code search approaches, including a version of IDCS (called there *interface-driven search*). We address several limitations of their study in our experiments. For example, in Hummel et al.'s study, each approach was applied with a different repository, which makes it difficult to conclude which approach performed better (it is very possible that the one applied with the largest repository would perform better). Moreover, the queries were generated by the authors themselves, not by independent subjects like in our experiment, which might have introduced bias to the evaluation. Query expansion is not addressed in the study either. In any case, results from the two studies are aligned in the sense that the approach that included types and keywords in their work was also the one that performed best.

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. [44] combine NLP and structural program analysis to locate and understand concerns in a software system. The basic difference from the AQE approach adopted in our paper 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 [44] requires user interaction, since similar terms have to be chosen by the user iteratively to construct the expanded queries. The AQE approach used in our paper is fully automatic and thus does not require user intervention. Gay et al. [45], on the other hand, 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 than the approach evaluated here.

Haiduc et al. [20] have recently proposed an approach for query reformulation based on machine learning to improve text-retrieval (TR) techniques. The difference from the approach used in this paper is that their technique is more general and suited to the context of TR-based concept location in source code. Moreover, it does not incorporate WordNet and does not handle types, as the approach evaluated here does. Sisman and Kak [46] 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 function implementations, and it does not deal with types either).

Most of the query reformulation approaches that were proposed in the past focused on concept location within a project – local-scale code search –, and mainly to identify code that must be dealt with in a particular maintenance task. The difference between such approaches and the one evaluated in our paper is that our goal is to help finding implementations of self-contained functions inside large open source code repositories, with the intent to reuse them. Moreover, several approaches require user intervention to be applied – *i.e.*, are not fully automatic – or do not deal with the specificities of IDCS (*e.g.*, do not handle types). As commented before, this is probably why the authors have identified NL lexicals to be ineffective for query expansion in their work.

Some recent approaches have proposed thesauri with word senses from a software-engineering context. For instance, Yang and Tan's [22] have proposed the identification of similar words in code by leveraging the context of words in comments and code. Tian et al. [47], on the other hand, have explored StackOverflow to automatically infer software-related similar words. The thesauri generated by such approaches could be used to further improve the AQE technique evaluated in this paper.

Our paper targets the comparison of a basic type of KBCS with IDCS, where the inclusion of return and parameter types is the primary constraint. Another interesting study would be to verify the effect of IDCS used along with more structured approaches to code search, such as V-DO pairs as implemented in the CONQUER system [48], or natural language phrases as proposed by Hill et al. [49].

VII. Conclusions

We have conducted an experiment to evaluate the use of types and query expansion within large-scale code search in the context of reusable Java functions. Our results point to essential findings in this topic. The main conclusions are the following: (1) that the inclusion of types in queries can significantly improve the recall of code search when used in combination with query expansion; (2) that query expansion performs best when types are included in queries, in comparison to when only keywords are used; and (3) that there are situations where the use of types should be avoided (*e.g.*, when keywords are semantically rich enough to describe the desired functions). These results are important in particular for mainstream code search tools: they suggest that developers of such tools should implement support for the inclusion of types in queries and query expansion (most of them do not support such features).

In-depth analysis of the *query successfulness* metric suggests that the best strategy to be used while performing code search is to first start with keywords only and then add types incrementally as satisfactory results fail to be returned initially. Our experiment also confirms the general suggestion about empirical studies that it is better to use more than a single metric while evaluating approaches, in order to reach more precise conclusions.

Future work includes further evaluation of other advanced approaches to code search, such as Test-Driven Code Search, in comparison to more traditional approaches. In this way, the body of knowledge pertaining code search can be increased, thus supporting better research and practice within the topic.

Acknowledgements

Otavio Lemos would like to thank FAPESP for financial suport (grant 2013/25356-2). The authors would also like to thank the subjects who participated in our experiment.

References

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

[2] 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.*, vol. 26, no. 2, pp. 80–87, Mar. 2009.

[3] D. Spinellis and C. Szyperski, "Guest editors' introduction: How is open source affecting software development?" *IEEE Softw.*, vol. 21, no. 1, pp. 28–33, Jan. 2004.

[4] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Comput. Surv.*, vol. 44, no. 1, pp. 1:1–1:50, Jan. 2012.

[5] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proc. of the ICSE 2011*. ACM, 2011, pp. 111–120.

[6] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 434–451, Jul. 2008.

[7] S. E. Sim, C. L. A. Clarke, and R. C. Holt, "Archetypal source code searches: A survey of software developers and maintainers," in *Proc. of the IWPC 1998*. IEEE Computer Society, 1998, p. 180.

[8] 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*. ACM, 2010, pp. 157–166.

[9] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, Jan. 2014.

[10] A. M. Zaremski and J. M. Wing, "Signature matching: A tool for using software libraries," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 2, pp. 146–170, Apr. 1995.

[11] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *Proc. of the ESEC/FSE 2005*. ACM, 2005, pp. 237–240.

[12] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proc. of the ASE 2007*. ACM, 2007, pp. 204–213.

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," in *Proc. of the PLDI 2005*. ACM, 2005, pp. 48–61.

[14] 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.*, vol. 53, pp. 294–306, April 2011.

[15] S. P. Reiss, "Semantics-based code search," in *Proc. of the ICSE 2009*. IEEE Computer Society, 2009, pp. 243–253.

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

[17] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 26:1–26:45, Jun. 2014.

[18] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proc. of the MSR 2014*. ACM, 2014, pp. 212–221.

[19] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Commun. ACM*, vol. 30, no. 11, pp. 964–971, Nov. 1987.

[20] 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*. IEEE Press, 2013, pp. 842–851.

[21] 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*. IEEE Computer Society, 2008, pp. 123–132.

[22] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Prof. of the MSR 2012*. IEEE, 2012, pp. 161–170.

[23] G. Little and R. C. Miller, "Keyword programming in java," in *Proc. of the ASE 2007*. ACM, 2007, pp. 84–93.

[24] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proc. of the CASCON 1997*. IBM Press, 1997, pp. 21–.

[25] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] G. A. Miller, "Wordnet: a lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[27] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proc. of the ICSE 2012*. IEEE Press, 2012, pp. 178–188.

[28] 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 2007*. ACM, 2007, pp. 13–22.

[29] 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*. IEEE Press, 2012, pp. 529–539.

[30] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 424–466, Aug. 2012.

[31] L. Martie and A. van der Höek, "Sameness: An experiment in code search," in *Proc. of the MSR 2015*. IEEE Press, 2015 (to appear).

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

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

[34] 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," Simula Research Laboratory, Tech. Rep. 6, June 2003.

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

[36] 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.*, vol. 28, pp. 721–734, 2002.

[37] M. D. McIlroy, "Mass produced software components," in *Proc. of NATO Softw. Eng. Conference*, P. Naur and B. Randell, Eds. Scientific Affairs Division, NATO, 1969, pp. 138–150.

[38] M. Rittri, "Using types as search keys in function libraries," in *Proc. of the FPCA 1989*. ACM, 1989, pp. 174–183.

[39] E. J. Rollins and J. M. Wing, "Specifications as search keys for software libraries," in *Proc. of the ICLP 1991*, 1991, pp. 173–187.

[40] A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 286–303, 1993.

[41] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proc. of the OOPSLA 2006*. ACM, 2006, pp. 413–430.

[42] M. Nurolahzade, R. Walker, and F. Maurer, "An assessment of test-driven reuse: Promises and pitfalls," in *Safe and Secure Software Reuse*, ser. Lecture Notes in Computer Science, J. Favaro and M. Morisio, Eds. Springer Berlin Heidelberg, 2013, vol. 7925, pp. 65–80.

[43] O. Hummel, W. Janjic, and C. Atkinson, "Evaluating the efficiency of retrieval methods for component repositories," in *Proc. of SEKE 2007*. KSI, 2007, pp. 404–409.

[44] 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*. ACM, 2007, pp. 212–224.

[45] 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*. IEEE, 2009, pp. 351–360.

[46] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proc. of the MSR 2013*. IEEE Press, 2013, pp. 309–318.

[47] Y. Tian, D. Lo, and J. Lawall, "SEWordSim: Software-specific word similarity database," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. ACM, 2014, pp. 568–571.

[48] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "Conquer: A tool for nl-based query refinement and contextualizing code search results," in *Proc. of ICSM 2013*. IEEE, 2013, pp. 512–515.

[49] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proc. of ICSE 2009*. IEEE, 2009, pp. 232–242.