Design and Evaluation of a Multi-Recommendation System for Local Code Search

Xi Ge[a], David C. Shepherd[b], Kostadin Damevski[c], Emerson Murphy-Hill[d]

[a]*Apple Inc., Cupertino, CA, USA*
[b]*ABB Corporate Research, Raleigh, NC, USA*
[c]*Department of Computer Science, Virginia Commonwealth University, Richmond, VA, USA*
[d]*Department of Computer Science, North Carolina State University, Raleigh, NC, USA*

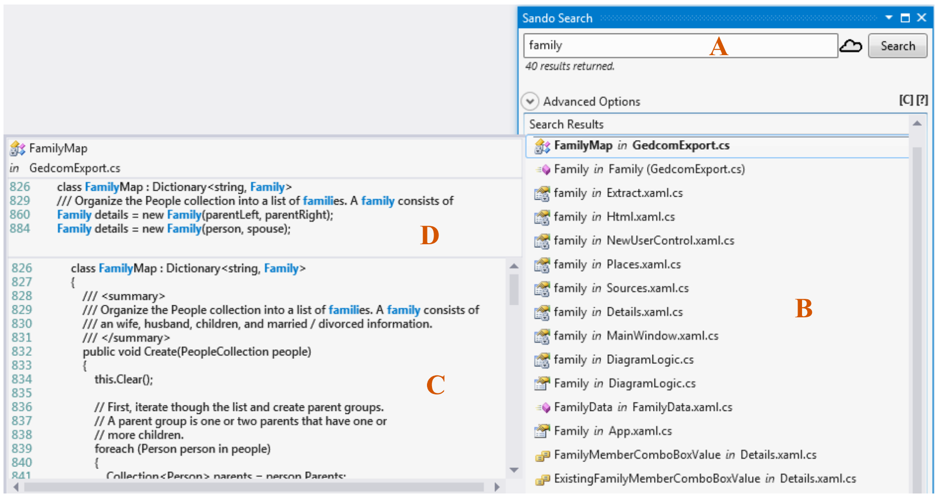
Abstract

Searching for relevant code in the local code base is a common activity during software maintenance. However, previous research indicates that 88% of manually-composed search queries retrieve no relevant results. One reason that many searches fail is existing search tools' dependence on string matching algorithms, which cannot find semantically-related code. To solve this problem by helping developers compose better queries, researchers have proposed numerous query recommendation techniques, relying on a variety of dictionaries and algorithms. However, few of these techniques are empirically evaluated by usage data from real-world developers. To fill this gap, we designed a multi-recommendation system that relies on the cooperation between several query recommendation techniques. We implemented and deployed this recommendation system within the Sando code search tool and conducted a longitudinal field study. Our study shows that over 34% of all queries were adopted from recommendation; and recommended queries retrieved results 11% more often than manual queries.

Keywords:

1. Introduction

Many programming tasks start with a search [1, 2]. However, searching code is difficult for developers: many are forced to use out-of-date tools, like regular expression based search tools, to search millions of lines of code they have never seen [3]. Since these tools depend solely on developers composing queries that exactly match unfamiliar strings, 88% of all queries using these tools return no relevant results [2]. Because developers spend as much as 40% of their time



```
family
40 results returned.
```

```
FamilyMap in GedcomExport.cs
Family in Family (GedcomExport.cs)
family in Extract.xaml.cs
family in Html.xaml.cs
family in NewUserControl.xaml.cs
family in Places.xaml.cs
family in Sources.xaml.cs
family in Details.xaml.cs
family in MainWindow.xaml.cs
family in DiagramLogic.cs
Family in DiagramLogic.cs
FamilyData in FamilyData.xaml.cs
Family in App.xaml.cs
FamilyMemberComboBoxValue in Details.xaml.cs
ExistingFamilyMemberComboBoxValue in Details.xaml.cs
```

```
FamilyMap
in  GedcomExport.cs
826    class FamilyMap : Dictionary<string, Family>
829    /// Organize the People collection into a list of families. A family consists of
860    Family details = new Family(parentLeft, parentRight);
884    Family details = new Family(person, spouse);
```

```
class FamilyMap : Dictionary<string, Family>
{
   /// <summary>
   /// Organize the People collection into a list of families. A family consists of
   /// an wife, husband, children, and married / divorced information.
   /// </summary>
   public void Create(PeopleCollection people)
   {
      this.Clear();

      // First, iterate though the list and create parent groups.
      // A parent group is one or two parents that have one or
      // more children.
      foreach (Person person in people)
      {
         Collection<Person> parents = person.Parents;
```

Figure 1: Sando screenshot.

searching, navigating, reading, and understanding source code [1, 2], these failed searches waste a lot of developers' time, potentially increasing the cost of software development.

Local code search tools help developers find a starting point for programming tasks. These tools are now distributed as part of popular integrated development environments (IDEs): InstaSearch [4] and Sando [5] are available for Eclipse [6] and Visual Studio [7], respectively. These tools take developers' queries as input and retrieve code elements from the currently opened project in the IDE. Researchers have shown local code search tools to be more effective than regular expression search tools [5], and have observed that such tools are used frequently, on a daily basis, by many developers in the field [8]. Unlike regular expression search tools like grep, local code search tools: (1) retrieve code elements, such as methods, classes and fields, instead of lines of text; (2) return ranked results, making more relevant results more accessible; and (3) index the codebase before developers need to search, making search almost instantaneous [5].

Despite these advantages, local code search tools suffer from some of the same issues that plague regular expression search tools. For instance, imagine a developer wants to search for how usernames are stored in an authentication system. They may search for "user file" or "user element". However, the implementation may refer to storage as a "user document". The developer would not think to compose this query without knowledge of the exact terms used in the codebase. Neither regular expression search tools nor local code search tools will help a developer in this scenario.

To address this problem, researchers propose multiple query recommendation techniques [9, 10, 11]. However, few of these techniques are empirically studied in the real-world setting. Without observing developers' actual interaction with the recommendation techniques, the usability and usefulness of these techniques are unclear. To fill this gap, we implemented Coronado, an extension to the Sando search tool that integrates various recommendation techniques to help developers compose new queries and refine failed ones. Taking advantage of the popularity of Sando, we were able to evaluate these recommendation techniques by collecting field data. In summary, the contributions of this paper are:

- An implementation of multiple query recommendation techniques as an extension to Sando, a local code search tool for Visual Studio [5], which we discuss in Section 2.

- The result of longitudinal field study, presented in Section 3, investigating Coronado's usability and usefulness. This study collected usage data from 591 unique Sando users for 24 months. Our study suggests that the queries recommended before manual search fails are equally useful to the recommendations afterwards; and recommending the identifiers in the codebase is the most effective technique.

This paper is an extension of our previous study of recommendations in code search [12]. Relative to the earlier work, this paper describes Coronado in greater depth. Also, by leveraging data collected from a significantly longer timespan than in [12], this paper strengthens the previous set of results by *analyzing data from more than two times as many users over more than three times as long a period of time*. A comparison of the results of the earlier field study from [12] to the one in this paper indicate consistent usage and effectiveness of the recommendation techniques.

The longer data collection timespan also allows for the selection of Coronado "super users", whose query recommendation use can be investigated over time, to determine if any change in developer recommendation use can be observed. Our results indicate that *recommendation use tends to increase over time*, as the developers likely gain confidence in this capability of the recommendation system to produce good queries, especially on unfamiliar code bases.

2. Sando Recommendations

Before delving into the recommendation techniques, we first introduce a local code search tool called Sando on which our techniques was implemented and evaluated. As a Visual Studio plug-in, Sando allows developers to search code snippets in their local codebase by issuing queries similar with those accepted by Google. Sando supports searches over C and C# code. Sando treats the different levels of software entities as documents, including classes, methods and fields. Search results are a list of documents that contain the terms in the input query. Sando orders the results based on Inverse Document Frequency scores that reflect a term's importance to a document in a collection of documents [13]. Figure 1 is a screen shot of Sando, where the search box is at the top (A) and the list view at the bottom (B) presents search results. Near the search box, the cloud button invokes a recommendation technique based on tag clouds as illustrated in Section 2.2, the "[C]" button clears the search history, and the "[?]" button provides help.

When a users single clicks one of the search results, Sando displays a pop up window to give her an overview of the corresponding code element. The lower half of the pop up window (C) has the entire code element, and the upper half contains the exact lines containing the queried terms (D). To open the file containing a search result, the user double clicks the result.

The design of Sando's recommendation system intertwines interactive query recommendation techniques, intended to aid the user in constructing an initial query, with semi-automatic query modification and expansion techniques, intended to help the user reformulate an initial query that has presumably yielded poor results. In this paper, we dub the first set of recommendations as pre-search and the second set, which occur after an initial query, as post-search recommendations. In both cases, Sando relies on user feedback to manually select, modify, or expand the query terms submitted to the system, in the spirit of a typical recommendation system. Such explicit user interaction is generally expected to produce more accurate results than a completely automated query expansion system, at the cost of more effort by the user.

We used Sando as our platform to implement and evaluate the recommendation techniques for the following reasons: (1) Sando is representative of local code search tools, as it implements similar functionality as the search tools in other integrated development environments; (2) Sando is an open source project with extensible APIs, so we can easily add features to it; (3) Users have downloaded Sando over fifteen thousand times[1], which provides us a significant number of users from which to collect feedback. We next detail the recommendation techniques in two steps. Section 2.1 describes the individual components used in our

[1]`goo.gl/jjuhP1`

techniques; Section 2.2 and Section 2.3 describe how we generate and present the recommended queries.

2.1. Components

To bridge the cognitive gap between local code search users and the codebase under search, we implemented an extension to Sando named Coronado that is based on five components: **codebase terms**, **term co-occurrence matrix**, **Verb-Direct-Object repository**, **software engineering thesaurus**, and **English thesaurus**. Before we explain how Coronado uses these components to recommend queries, we first discuss each component and how Coronado collects and maintains those contents.

2.1.1. Codebase terms

The first component of Coronado contains the terms in the codebase under search. Coronado collects these terms when Sando's indexer traverses the programmer's project. Sando's indexing process breaks a source code element into a set of terms, including raw identifier names and terms extracted by splitting identifiers that use camel case. For instance, Sando indexes a method name "parseFile" as three terms: "parse", "file", and "parseFile". C and C# keywords are not indexed.

Sando indexes the codebase in two stages: the initial indexing and the incremental indexing. Sando performs the initial indexing only when the developer opens a project whose index information has not been cached previously by Sando. Sando performs incremental indexing whenever the developer changes a project whose index information is cached. Initially indexing a project of $10,000$ lines of code takes Sando about twenty seconds, and incrementally indexing a changed C# file takes about twenty milliseconds. By reusing the terms collected during indexing, Coronado builds and maintains the set of terms in the codebase under search.

2.1.2. Term co-occurrence matrix

The second component of Coronado is a matrix that records the number of co-occurrences of every two terms that appear together. For each pair of terms $[t_1, t_2]$ that occur together in a source document, the matrix builder increments the current value of the element at $[t_1, t_2]$ in the matrix by one. To give an example, consider the following code snippet:

```
PathManager CreatePathManager(string path) {
  if(Path.HasExtension(path))
    return new PathManager(Path.GetDirectoryName(path));
  else
    return new PathManager(path);
}
```

The method contains the following twelve terms: *path*, *manager*, *PathManager*, *create*, *CreatePathManager*, *has*, *extension*, *HasExtension*, *get*, *directory*, *name*, and *GetDirectoryName*. Therefore, for this method, the co-occurring pairs are the combinations of any two terms; that is, 144 ($12 * 12$) in total. When the developer adds this method to the codebase, the term co-occurrence matrix increments the current value of each element corresponding to each pair by one, for example, elements of $\{$*create, path*$\}$, $\{$*path, manager*$\}$ and so on.

$$\begin{array}{cc} & \begin{array}{ccc} \text{“blue”} & \text{“green”} & \text{“red”} \end{array} \\ \begin{array}{c} \text{“blue”} \\ \text{“green”} \\ \text{“red”} \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \end{array} \longleftrightarrow \begin{array}{l} \text{Row = \{ “blue”, “green”, “red”\}} \\ \text{Column = \{“blue”, “green”, “red”\}} \\ \text{A = \{1, 1, 2, 1, 1, 2, 1\}} \\ \text{IA = \{0, 3, 5\}} \\ \text{JA = \{0, 1, 2, 0, 1, 0, 2\}} \end{array}$$

Figure 2: Yale format for a sparse matrix.

One caveat is that keeping the co-occurrence matrix in memory is inefficient. For instance, consider Sando itself, a project with about 10,000 lines of code and about 2000 terms. For such a project, the number of elements in the matrix is about 4 million ($2000 * 2000$). To improve memory efficiency, we exploited the fact that the co-occurrence matrix is usually sparse; that is, according to our tool usage data, over 90% of the elements in the matrix contain the value of 0. Thus, we represent the co-occurrence matrix by using the Yale format, a data structure that efficiently stores sparse matrices without substantially impacting lookup and insertion speed [14].

The Yale format represents a sparse matrix by using three arrays: (1) the values of all non-zero elements in the matrix are stored in row-major form; (2) the indices of the first non-zero elements of the matrix's rows (1); and (3) the column indices of the elements in array (1). Figure 2 gives an example of Yale format where A, IA and JA correspond to the aforementioned three arrays respectively. The more zeros in the matrix, the more the Yale format improves memory efficiency compared to a normal matrix.

2.1.3. Verb-Direct-Object repository

Another component in our Coronado technique is Verb-Direct-Object repository. Based on the observation that source code is often about performing actions on objects, Fry and colleagues proposed a technique that combines natural language processing and source code analysis [15]. Their technique extracts Verb-Direct-Object pairs from a codebase, such as "open file" or "close stream". By adopting this technique, Coronado periodically analyzes the codebase under search, extracting the Verb-Direct-Object pairs and caching them for future recommendations.

2.1.4. Software engineering thesaurus

Coronado also uses a thesaurus of terms that are frequently used in software development. Software engineering has developed its own set of terms, synonyms and abbreviations that do not exist in conventional English, such as "imap". Some software engineering terms have specific meanings that are different than their conventional English meanings, such as "class". Thus, simply reusing a general thesaurus cannot help interpret field-specific information, and can even be detrimental to client software tools [16]. To build a field-specific thesaurus, we reuse Gupta and colleagues' work that mines the relationship between different terms in source code and generates 1724 pairs of semantically related terms. Among these pairs, 91% are synonyms specific to the field of software engineering [16]. The examples of these field-specific synonyms include "execute"–"invoke", "load"–"initialize" and "instantiate"–"create". In addition to the related terms, their work also quantifies the commonality of the pairs of synonyms. For instance, "create" has been found to be related to several terms, including "make", "do", and "construct". However, "make" is more closely related to "create" than it is to "construct" or "do".

2.1.5. English thesaurus

The final component of Coronado is a general English thesaurus. To build this thesaurus, we reused Miller's lexical database WordNet [17]. WordNet represents related concepts as a graph, where nodes are words and synonym edges connect words with similar meanings. Thus, finding synonyms using WordNet takes constant time because Coronado simply looks for neighbors of a given word. Although WordNet obtains a complete set of English words, keeping the whole data set in memory is costly. Hence, we reduced the size of WordNet to include the $100k$ most frequently used words in English [18]. We speculate that the $100k$

words are sufficient for most queries, although how often a user queries with unusual words remains an open question because, for privacy reasons, we do not collect data about what terms Sando users are searching for.

2.2. Pre-search recommendation

We next describe how Coronado uses these components. Whenever a developer types something into Sando's search box, Coronado suggests queries even before the developer clicks the search button as illustrated in Figure 3. We call these *pre-search recommendations*. These recommendations come from three sources: identifiers, Verb-Direct-Object pairs, and co-occurring terms.

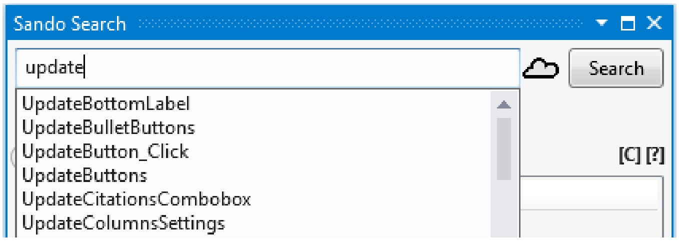
Since our pre-search recommendations are displayed to the user at every key press in the search box, the response time for their lookup is important to their usability. We measured pre-search recommendation generation response times on the order of seconds for large codebases (e.g. the Linux kernel), which definitely affected Sando's usability. Like many other search tools, we found that pre-generating a trie data structure [19] from the recommendation terms reduces response time by orders of magnitude at the cost of relatively little additional memory and CPU time.

2.2.1. Identifiers

The first source of the pre-search recommendations are identifiers. When the programmer types a string into the search box, Coronado looks through the codebase terms component to determine whether that string is a prefix of any identifier in the local terms. Each identifier is then recommended to complete the developer's search string, and displayed in a drop-down menu below the search box. For example, Figure 3a displays what search terms are recommended to the developer when she types the string "update" when searching over the Family.Show codebase, an open source project that allows users to build family trees [20].

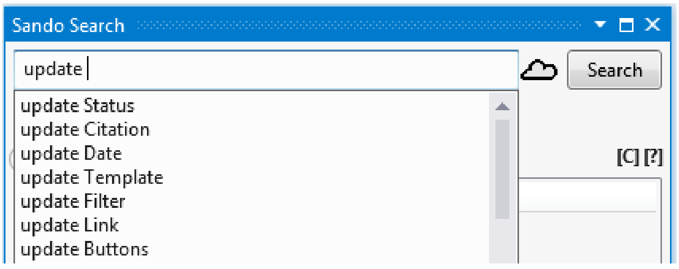
2.2.2. Verb-Direct-Object pairs

When the developer hits spacebar after typing a verb in the search box, Sando retrieves the Verb-Direct-Object pairs whose verb is the given search term, from the Verb-Direct-Object repository component. If Sando finds multiple Verb-Direct-Object pairs, Sando ranks them by using the co-occurrence matrix; the more often the verb and direct object appear together, the higher up the pair will appear in the drop-down box. The drop down menu in Figure 3b exemplifies the Verb-Direct-Object recommendations when the developer inputs "update" to the search box; because update co-occurs most often with Status than any of the other objects in



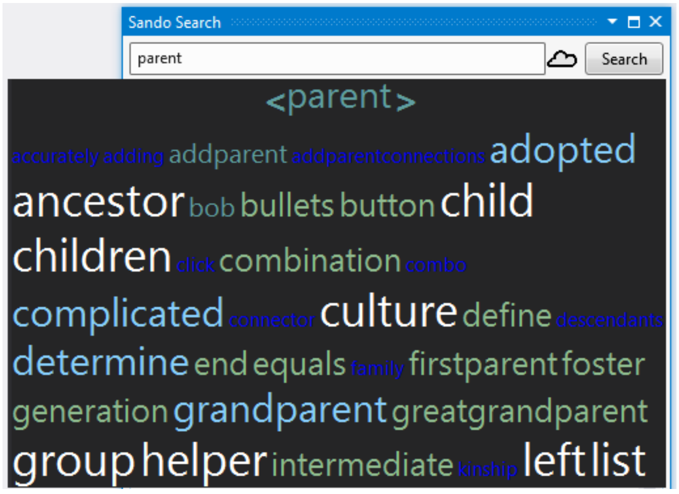
```
UpdateBottomLabel
UpdateBulletButtons
UpdateButton_Click
UpdateButtons
UpdateCitationsCombobox
UpdateColumnsSettings
```

(a) Recommending identifiers.



```
update Status
update Citation
update Date
update Template
update Filter
update Link
update Buttons
```

(b) Recommending Verb-Direct-Object pairs.


Sando Search
parent
Search
<parent>
accurately adding addparent addparentconnections adopted
ancestor bob bullets button child
children click combination combo
complicated connector culture define descendants
determine end equals family firstparent foster
generation grandparent greatgrandparent
group helper intermediate kinship left list

(c) Frequently co-occurring terms.

Figure 3: UI of pre-search recommendations.

the Verb-Direct-Object list in Family.Show, “update Status” is the most highly recommended search query.

2.2.3. Frequently co-occurring terms

The third type of pre-search recommendation suggests frequently co-occurring terms by using the term co-occurrence matrix. After inputting one or more terms in the search box, the developer can click the cloud button near the search box to show a tag cloud. The terms in the tag cloud are those that co-occur most frequently with the term or terms in the search box. The bigger the font size in the tag cloud, the more co-occurrences the term has with the terms in the search box. Figure 3c shows an example – this tag cloud presents terms that co-occur frequently with "parent" in Family.Show. The term "children" co-occurs more frequently with "parent" than "define" does. Clicking on a term in the tag cloud adds that term to the end of the original search query.

For frequently co-occurring terms, we chose to visualize the terms in a tag cloud, rather than a conventional drop-down list, like we used for identifier and Verb-Direct-Object recommendations. We made this choice because our early experiments suggested that for a given codebase, there are more co-occurring terms than either identifiers or Verb-Direct-Object pairs. Thus, because tag clouds use screen space more efficiently than drop-down lists, we used them for recommending term co-occurrence. In later discussion, we also refer the frequently co-occurring terms as the tag cloud recommendation.

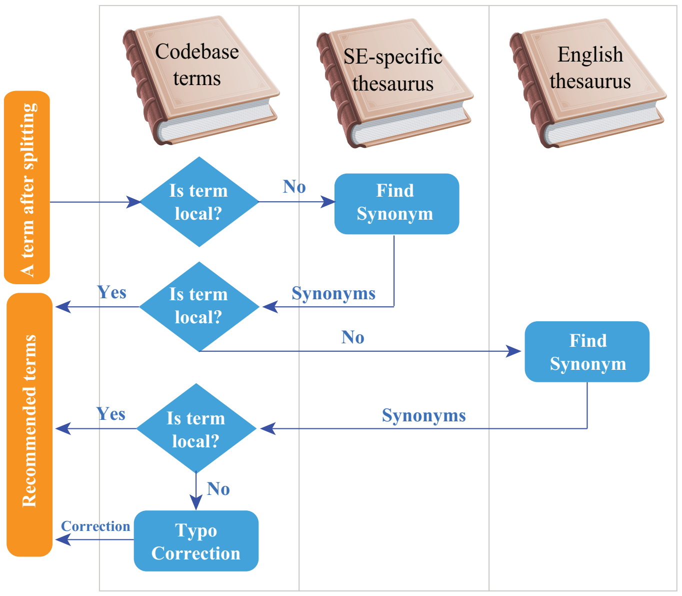


Figure 4: Post-search recommendation.

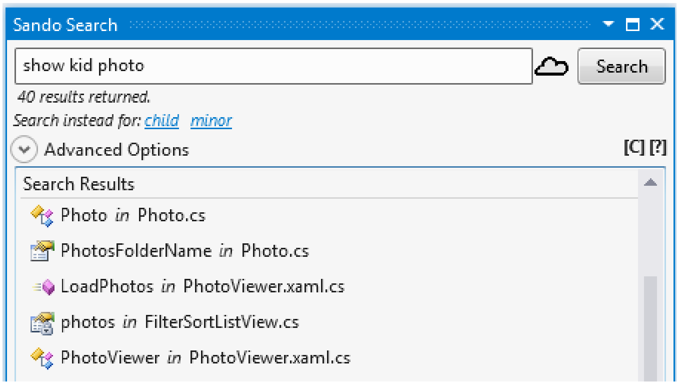
2.3. *Post-search recommendation*

In addition to helping the developer complete queries, Coronado also issues recommendations after the developer clicks the search button and no search results were returned. In this case, the developer's query contains terms that do not exist in the codebase. To help her compose a new query, Coronado uses the codebase terms, software engineering thesaurus, and English thesaurus to recommend semantically similar terms in the codebase under search. Supposing the original query has several space-separated terms, Coronado first queries the codebase terms component to check whether each term exists in the codebase. If a term does not exist in the codebase, and thus the search fails, Coronado uses the following three steps to generate a new query.

2.3.1. *Pre-processing*

If a term does not exist in the codebase, some parts of the term might. Thus, Coronado greedily splits the term and incorporates the parts that exist in the local terms to the recommended queries. More specifically, for a given term in the query, Coronado tries to find its longest prefix and suffix that exist in the local terms, and in turn, the codebase itself. Next, the technique recursively splits the middle part of the term until no in-codebase prefix or suffix can be found.

For example, suppose the developer searches for "getelementname", which does not exist in the codebase. If the local terms contain "get" and "name", splitting the searched term leads to three new terms: "get", "element", and "name".



show kid photo

40 results returned.

Search instead for: child minor

Photo *in* Photo.cs

PhotosFolderName *in* Photo.cs

LoadPhotos *in* PhotoViewer.xaml.cs

photos *in* FilterSortListView.cs

PhotoViewer *in* PhotoViewer.xaml.cs

Figure 5: UI of post-search recommendations.

2.3.2. Synonym recommendation

After splitting, if the split terms do not exist in the local terms, Coronado tries to find synonyms for them in the thesauri. Coronado starts with the software engineering thesaurus due to its higher relevance than the English thesaurus. If no synonyms in the software engineering thesaurus are found, Coronado tries to find synonyms in the English thesaurus. After retrieving the synonyms, the technique next excludes those synonyms that are not in the local terms. The remaining synonyms are recommended to the developer as the replacements to the original term. If finding multiple synonyms to recommend, Coronado uses the term co-occurrence matrix stated in Section 2.1 to rank them. The synonyms that occur more frequently with the other terms of the input query rank higher. We illustrate the process of finding synonyms in Figure 4.

2.3.3. Typo correction

For those terms that neither exist in the local terms by themselves nor have synonyms in the thesauri, Coronado considers them as the typos of a term in the local terms. Therefore, Coronado uses these terms to correct them. The algorithm for correction adopts 2-gram indexing to quickly find the terms that spell similarly with a given typo [21].

More specifically, this correction algorithm first creates a correction table with 416 $(26 * 26)$ rows where each row is labeled by a pair of letters (from “aa”, “ab”, “ac” to “zz”). Next, for every term in the local terms, Coronado inserts the term to those rows in the table whose label is a part of the term. For instance, the term “example”, assuming it is in the local terms, is inserted to the rows of “ex”, “xa”, “am”, “mp”, “pl” and “le”. After inserting all of the local terms, Coronado is ready to use this table to correct a given typo by going through the following steps:

- For the typo, we first calculate the rows it will be inserted into as if the typo were a term.
- We next analyze the existing terms in these rows, and find out the term that shares the most common rows with the typo as its correction. If finding multiple such terms, we further select the one whose edit distance to the typo is the smallest as the correction to the typo.

After calculating the post-search recommendations, Sando displays the recommended queries under the search box; each query is a hyperlink, the click on which leads to Sando’s searching the corresponding query, as illustrated in Figure 5.

3. Field Study

We conducted a longitudinal study to answer the following research questions:

- RQ1: How often do developers use Coronado's recommendations during their normal work to improve their queries?
- RQ2: Do developers seem satisfied with queries constructed from Coronado's recommendations?
- RQ3: Do Coronado's recommendations remain relevant to users over time?

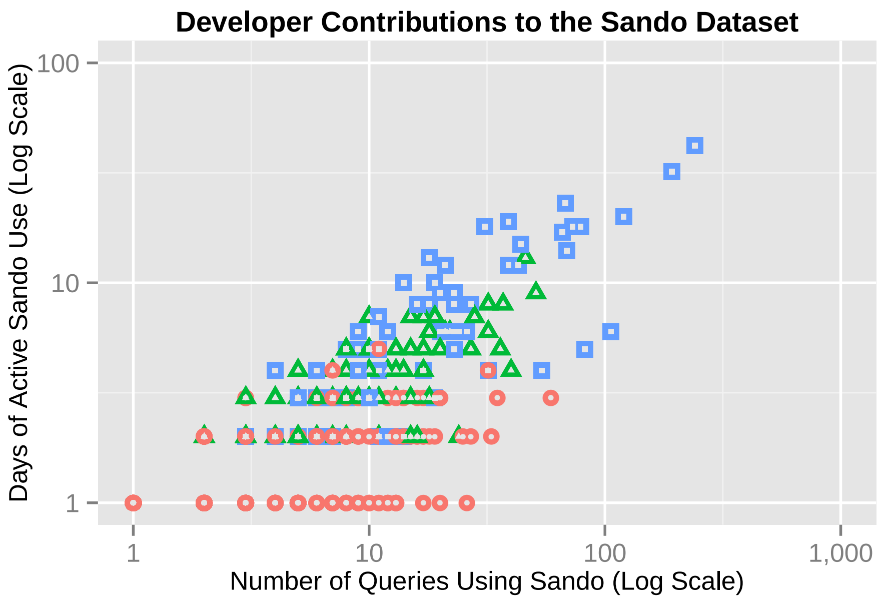


Figure 6: A plot of the characteristics of the Sando usage data, collected from 591 developers in the wild, showing the number of queries for each user on the X axis, and the number of days developers interacted with Sando on Y axis. A ○ denotes that a user was active for less than one week, △ denotes Sando interactions of one week to one month, and □ denotes activity spanning over one month.

Table 1: Use of query recommendations in Sando users' data.

Interaction Type	Number of Queries
Total Queries	4917
Manual Queries	3275 (66%)
Recommended Queries	1642 (34%)
Passive Rec.	969 (59%)
Active Rec.	673 (41%)
Lookup Intent	441 (66%)
Exploratory Intent	233 (34%)

3.1. Study Setting

We evaluated Coronado through a collection of anonymous usage data from Sando users in the wild. Sando automatically uploads usage data to S3 cloud storage [22] for users that give their permission. Coronado was instrumented to log two types of events, recommendation *review* and recommendation *acceptance*. More specifically, the events that we included in the log are:

- *The developer reviews a pre-search recommendation.* An event is written to the log when a user reviews a pre-search recommendation, either by highlighting a drop-down recommendation (i.e., highlighting a recommendation in Figure 3a) or displaying a tag cloud (Figure 3c).

- *The developer accepts a pre-search recommendation.* An event is written to the log when a developer accepts a pre-search recommendation, either by choosing a recommendation from the drop-down list or by clicking on a term in the tag cloud.

- *A user reviews post-search recommendations.* Each time a Sando user submits a query containing terms that do not exist in the codebase under search, Sando issues a set of post-search recommendations, displayed as clickable links (see Figure 5). Each time Sando displays these links, an event is logged.

- *The developer accepts a post-search recommendation.* After Sando issues post-search recommendations, the user may accept these recommendations by clicking on the hyperlink. Each time a link is clicked an event is written to the log.

In addition to the aforementioned events, Sando also logs events that apply to all queries. These events include:

- *The developer submits a query.* Each time a user searches using Sando, we log the number of results returned.
- *The developer examines a search result.* Sando provides two ways to examine a search result, via a preview pop-up summary of the result or by opening the relevant file in the code editor. We log both of these events.

3.2. Results

We released a version of Sando with Coronado recommendations to the Visual Studio Gallery site [23] and continuously gathered usage data for approximately 24 months. In total, we collected logs submitted by 591 unique Sando users[2].

As shown in Figure 6, the majority of the users issued few queries, while several power-users issued upward of 100 queries using the tool. There was a steady stream of queries gathered over the time period and the dataset was not dominated by queries gathered over a short time span. As more developers downloaded Sando over the collection period, the number of collected queries steadily increased from month to month. Each developer issued a range of between 1 and 241 Sando queries, while the mean number of queries per developer was 8.49 with a standard deviation of 17.22 queries, indicating that the data set was also not influenced by a small group of developers.

All known Sando and the SrcML.NET developers were excluded from the data set (SrcML.NET is a service used by Sando). We also excluded Sando users that generated only one query with the tool, under the assumption that they were only trying the tool and were not using it to perform any maintenance activity. In the following analysis, we used data from the remaining 487 users that issued 4917 queries.

RQ1. Use of recommendations. During the collection period, Sando users executed 4917 queries, 1642 of which utilized recommendations (34%), as shown in Table 1. We observe that the recommendation usage rate is fairly high, considering that developers are working on their own systems, that they are likely not accustomed to code search tools with recommendations, and that there was no in-tool documentation of the recommendation feature.

[2]Data for a 25 day period, consisting of 44 separate users issuing a combined total of 363 queries, is available at `https://github.com/abb-iss/SandoRecommendationStudyData/`

Both the adoption rate of recommendations, used in about 34% of all queries, and the raw number of recommendations utilized, 1642, indicate that recommendations are having a real impact on developers in the field.

We categorized the observed recommendations into two basic categories, pre-search and post-search. Developers used pre-search recommendations 673 times while they employed post-search recommendations 969 times (see Table 1). This indicates that developers actively sought recommendation usage in 41% of the recommended queries while they accepted recommendations after the query in 59% of the cases.

The balance between pre-search and post-search recommendations suggests that both types of recommendations in Coronado are equally useful.

Of the pre-search recommendations (673 in total) 112 were Verb-Direct-Object pairs, 441 were identifiers, and 115 were from tag clouds. Heavy use of identifier recommendations is an indication of searches for the purpose of information lookup, where the developer likely has a specific part of the code (e.g. a specific method or class name) in mind. On the other hand, using the Verb-Direct-Object pairs and tag cloud recommendations are likely to be used in more exploratory searches, as exemplified in Figure 6(c), where the developer only vaguely knows what she is looking for.

The distribution of recommendation types suggests that a substantial part (34%) of recommendations usage in the field is during exploratory searches.

Table 2: Failure rates for Sando queries.

Recommendation Type	Failure Rate
Manual (No Rec.)	49%
Recommended	38%
Verb-DO	21%
Identifier	19%
Tag Cloud	73%

RQ2. Quality of search results for recommended queries.

When investigating user satisfaction with a search query it is valuable to measure cases where searches completely fail, as they are a straightforward indicator of the inadequacy of the result set returned by the search tool. A large number of inadequate result sets translates to anticipated dissatisfaction of the users with the code search tool. In our data, we considered a search to have failed if there was no further interaction with the search UI (e.g., no clicks on any kind, single or double, on the results) after the query. During our collection period, of the 1641 queries from recommendations 623 resulted in failure (38%) and of the 3275 manual queries 1611 resulted in failure (49%), as shown in Table 2.

Recommendation-based queries received a click more often than manual queries.

Using failed queries as our metric, we can examine the effectiveness of each of the recommendation techniques that constitute our Coronado approach. Of the 112 queries where the users relied on the Verb-Direct-Object recommendation, only 23 (or 21%) were unclicked (i.e. failed) queries. Similarly, for identifier recommendations, 84 out of 441 (or 19%) queries failed. Tag cloud recommendations fared a lot worse, 84 out of 115 queries (or 73%) yielding no interactions with the result set. The post-search recommendation techniques in Coronado yielded a success rate of 43% (419 failed queries out of 969 issued), matching the success rate of manual queries.

Identifier and Verb-Direct-Object recommendations were most successful when used by developers in the field.

RQ3. Recommendation use on the same code base over time.

Our final research question evaluates whether users rely more on recommendations over time, as, presumably, they become more familiar with Coronado. To measure this, we extracted sequences of 20 or more queries on the same code base by the same user, and considered recommendation use in the first 10, relative to the last 10 queries.

There were 24 query sequences that satisfied this condition in the Sando dataset. Descriptively, during the first 10 queries users used a mean of 3.29 recommendations, compared to 5.16 recommendations during the last 10 queries. We used statistical hypothesis testing, using the one-tailed Wilcoxon signed-rank test to determine whether the increase observed in the means of the first and latter set of queries was significantly different. Based on the computed p-value of 0.00013, which is less than the 0.05 significance level, we reject the null hypothesis that the recommendation use frequency in the two sequence intervals is identical. Instead, we adopt the alternative hypothesis that the latter queries use more recommendations than the earlier set.

Recommendation use on the same code base by the same user over time tends to increase.

3.3. Differences to Prior Results and Analyses

Coronado was initially presented in [12], coupled with initial results for RQ1 and RQ2. In this paper, analysis of those two research question is conducted with a dataset of nearly twice as many queries (4917 vs. 2563 in [12]) and more than twice as many users (591 vs. 274 in [12]). The results for RQ1 and RQ2 remained very consistent with the ones reported in the original study, as follows. The use of recommendation remained steady, changing only 2 percentage points from 32% to 34% on the larger dataset. The use of active recommendations was similar 41% compared to 42% in [12]. The failure rates also remained generally consistent in the larger datasets: 42% of manual queries in the smaller to 49% in the larger dataset; 35% of recommended queries to 38% in the current, larger dataset; 19% of Verb-DO recommended queries to 21% in the larger dataset; and, exactly the same, at 19%, for identifier recommended queries. The fact that the results remained, qualitatively, the same, while the number of queries, users, and length of time increased significantly, to 24 months of data collection, further underscores the validity of the results reported for RQ1 and RQ2.

To better understand the variance of our results, we performed bootstrapping (i.e. random resampling with replacement) on the data from the larger dataset. We used 1000 random samples of 10,000 queries randomly sampled from the dataset. We generally found low variances across all of the measures. For instance, at 99% confidence, based on this technique, the confidence interval of the failure rate of manual queries was (47.6% - 50.8%), while for recommended queries the interval was (38.6% - 42.1%). The failure rate of the Verb-DO recommendations was in the range (13.6% - 27.4%), and for identifier recommendations (15.6% - 22.4%), exhibiting larger ranges due to relatively less data of each recommendation type.

3.4. Implications

Based on the data from the longitudinal study, several implications may help researchers improve the usability and the usefulness of query recommendation techniques for code search. We next summarize these implications.

Both preventive and the corrective recommendations are important. As the results of RQ1 suggest, pre-search and post-search recommendations are equally useful, suggesting that developers need both reminders and corrections to compose effective queries. Existing query recommendation techniques integrates no feedback loop from the retrieved results, thus they cannot correct the failing queries [9, 10].

Simply recommending identifiers in the codebase can reasonably assist developers. Recommending the identifiers in codebase may be the most straightforward technique we implemented. However, according to the collected data, developers adopted this recommendation more frequently than more sophisticated ones, suggesting that the basic program entity serves as an important clue for developers' retrieval of the relevant code snippets.

Verb-Direct-Object pairs are effective for exploratory searches. The low failure rate and significant adoption rate of Verb-Direct-Object pairs are especially encouraging as these recommendations, unlike identifier recommendations, occur on searches where the developer is unfamiliar with (a segment of) the codebase and does not have a specific program element in mind. Such exploratory searches can be more challenging for a code search technique and therefore the low measured failure rate for these queries should be considered a successful outcome for this technique.

The readability of the recommended queries matters. Different recommendation techniques lead to the varied readability of the recommended queries. For instance, aligning with how developers explain code snippets, the recommended identifiers and the Verb-Direct-Object pairs are more readable and understandable than the queries recommended from the other techniques, which difference explains the former two's higher adoption rate. As another example, the tag cloud recommendation, despite of effectively capturing the related terms in the codebase, attracts less adoption due to the distance between the fragmented information and the developers' perception of the codebase.

Recommending queries followed by manual selection leads to more useful results than using either one alone. Existing query expansion techniques silently add terms to developers' original queries without explicitly interacting with the developers [9, 10, 11]. Our study shows that developers' explicit selection improves the usefulness of the recommended queries, leading to more promising results retrieved.

Developers trust recommendations more over time. As developers experience the benefits of a query recommendation system, they tend to utilize it more over time, likely relying on the suggestions of the system to avoid failed searches. Our study shows that this trend exists over a relatively small number of queries, for relatively few users.

3.5. Threats to validity

Our log file analysis is potentially susceptible to several threats, both internal and external. One internal threat is how we measure user satisfaction of retrieved results. Conventional measurements of the quality of the retrieved results, such as precision and recall [24], cannot be collected by analyzing the log files. However, failed queries, or the clickthrough rate, which is the name for this metric in the Internet search community has consistently been shown to produce a reliable measure [25], albeit somewhat coarse-grained, of user satisfaction with a result set.

Externally, we collected data only from 591 users for a period of about 24 months. The results drew from the limited number of users and time length may be not generalisable to other Sando users for a longer period of time. Another external threat is that we investigated only several recommendation techniques, thus the observed usefulness and usability may not apply to other techniques. In future, we plan to add more recommendation techniques to Sando and continuously monitor the users' interaction with them.

4. Related Work

A large body of existing work is related to ours. In this section, we summarize them from two different themes, namely query reformulation and improving code search.

Query reformulation. In information retrieval systems, queries of higher quality can lead to more relevant results. The original queries provided by users may not be good enough; hence researchers proposed multiple ways to reformulate them. In general, query reformulation techniques either expand [9], or reduce [10], given queries. Specific to the field of software engineering, Haiduc and colleagues proposed a recommender called Refoqus that suggests query expansion by analyzing a set of attributes of given queries [26]. Semi-automatic query reformulation techniques integrate developers' feedback to improve the original queries. For example, De Lucia and colleagues proposed a technique that uses developers' feedback to incrementally discover traceability [27]. Coronado differs from all these techniques in that Coronado recommends related search terms and refine failed ones; that Coronado guarantees that the recommended terms lead to some results; and that Coronado uses software engineering-specific lexical information to find related terms.

Improving Code Search. To better assist developers retrieve relevant code snippets, researchers proposed various techniques. For instance, Yang and Tan leverage the context of words to mine semantically related terms in the codebase [28]. Ali and colleagues combine software repository mining results with information retrieval techniques to improve the accuracy of the later [29]. Sisman and Kak enrich developers' queries by injecting terms appearing in the artifacts drawn from their manual queries [11]. Bajracharya and colleagues proposed Structural Semantic Indexing that associates code snippets by the APIs they used, thus the developers can easily retrieve the usage examples of certain APIs [30]. Marcus and Maletic apply latent semantic indexing to recover the traceability link between documentation and source code [31]. Different from these works, this paper aims at evaluating the usefulness and the usability of existing techniques instead of proposing new ones.

5. Future Work

Future research can build on our work by improving and refining search recommendation techniques. For instance, future search tools could apply the degree-of-interest model to better rank recommended queries [32], using contextual information to promote search terms related to developers' recent activity.

The second way to improve the recommendation technique is to integrate domain-specific lexical knowledge for the software under search. In the current implementation of Coronado, we only used the software engineering thesaurus and the English thesaurus to recommend related terms after the developer's manual search fails. We believe a domain-specific thesaurus can further improve the quality of recommended terms. For instance, if a developer searches in a medical application, recommending related terms in the medical domain potentially leads to more promising queries.

Another way to improve the state of the art in recommendation techniques is to suggest queries to developers based on their colleagues' successful queries. Such a technique could leverage collaborative filtering using data from developers who work on the same codebase to recommend better and more relevant queries [33].

Before developing better recommendation techniques, conducting empirical studies of local code search tool users is helpful. Analyzing log files, as we did in our field study, can show usage patterns, but cannot uncover the causes of developers' behavior. For instance, we know that users use pre-search recommendations less often than post-search recommendations, but we do not know the causes without a user study.

Another area of further inquiry could be in ways to more rapidly build trust in the recommendation system, in order to increase its rate of usage and reduce failed queries. Several means for achieving this goal are possible, such as improving the tool's UI, creating in-tool suggestions or demonstrations, or providing visible external learning materials.

6. Conclusion

Local code search tools help developers easily find the code they want to reuse or edit. However, composing the right queries can be difficult for developers who are not familiar with the codebase under search. To help developers compose queries that return relevant parts of the codebase, researchers proposed various query recommendation techniques. To investigate the usability and the usefulness of these recommendation techniques, we integrated several of them to the Sando search tool and conducted a longitudinal field study. We found that, in the field, developers issued 34% of all their queries by taking recommendations; preventive and corrective recommendations are equally useful; and that developers tend to adopt recommendations that are more readable.

Acknowledgment

This research was conducted during the first author's internship at ABB Research. We thank the participants for helping us conduct the experiments. We thank the help from Vinay Augustine, Patrick Francis, and Will Snipes. We also thank the comments from the Development Liberation Front group members Titus Barik, Michael Bazik, Brittany Johnson, Kevin Lubick, John Majikes, Yoonki Song and Jim Witschey.

[1] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil, An examination of software engineering work practices, in: Proceedings of the conference of the Centre for Advanced Studies on Collaborative research, 1997, pp. 21–36.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on Software Engineering 32 (12) (2006) 971–987.

[3] J. Sillito, G. C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the international symposium on Foundations of software engineering, 2006, pp. 23–34.

[4] InstaSearch - Eclipse plug-in for quick code search, https://code.google.com/a/eclipselabs.org/p/instasearch/ (2013).

[5] D. Shepherd, K. Damevski, B. Ropski, T. Fritz, Sando: an extensible local code search framework, in: Proceedings of the International Symposium on the Foundations of Software Engineering, 2012, pp. 15:1–15:2.

[6] The Eclipse Foundation, http://www.eclipse.org/ (2013).

[7] Microsoft Visual Studio, http://www.microsoft.com/visualstudio/ (2013).

[8] K. Damevski, D. Shepherd, L. Pollock, A field study of how developers locate features in source code, Empirical Software Engineering (2015) 1–24doi:10.1007/s10664-015-9373-9.
URL http://dx.doi.org/10.1007/s10664-015-9373-9

[9] C. Carpineto, G. Romano, A survey of automatic query expansion in information retrieval, ACM Computing Surveys 44 (1) (2012) 1:1–1:50.

[10] N. Balasubramanian, G. Kumaran, V. R. Carvalho, Exploring reductions for long web queries, in: Proceedings of the international conference on Research and development in information retrieval, 2010, pp. 571–578.

[11] B. Sisman, A. C. Kak, Assisting code search with automatic query reformulation for bug localization, in: Proceedings of the Working Conference on Mining Software Repositories, 2013, pp. 309–318.

[12] X. Ge, D. Shepherd, K. Damevski, E. Murphy-Hill, How developers use multi-recommendation system in local code search, in: Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on, IEEE, 2014, pp. 69–76.

[13] H. C. Wu, R. W. P. Luk, K. F. Wong, K. L. Kwok, Interpreting tf-idf term weights as making relevance decisions, ACM Transactions on Information Systems 26 (3) (2008) 13:1–13:37.

[14] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, A. H. Sherman, Yale sparse matrix package i: The symmetric codes, International Journal of Numerical Methods in Engineering 18 (1982) 1145–1151.

[15] Z. Fry, D. Shepherd, E. Hill, L. Pollock, K. Vijay-Shanker, Analysing source code: looking for useful verb-direct object pairs in all the right places, IET Software 2 (1) (2008) 27–36.

[16] S. Gupta, S. Malik, L. Pollock, K. Vijay-Shanker, Part-of-speech tagging of program identifiers for improved text-based software engineering tool, in: Proceedings of the International Conference on Program Comprehension, 2013.

[17] G. A. Miller, Wordnet: A lexical database for english, Communications of the ACM 38 (1995) 39–41.

[18] Frequent Word Lists, http://invokeit.wordpress.com/frequency-word-lists/ (2013).

[19] P. Brass, Advanced Data Structures, Cambridge books online, Cambridge University Press, 2008.
URL `https://books.google.com/books?id=g8rZoSKLbhwC`

[20] Family.Show open source project, https://familyshow.codeplex.com/ (2013).

[21] K. Kukich, Techniques for automatically correcting words in text, ACM Computing Surveys 24 (4) (1992) 377–439.

[22] Amazon S3 Cloud Storage, http://aws.amazon.com/s3/ (2013).

[23] Visual Studio Gallery, http://visualstudiogallery.msdn.microsoft.com (2013).

[24] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.

[25] M. Richardson, E. Dominowska, R. Ragno, Predicting clicks: Estimating the click-through rate for new ads, in: Proceedings of the International Conference on World Wide Web, 2007, pp. 521–530.

[26] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, T. Menzies, Automatic query reformulations for text retrieval in software engineering, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 842–851.

[27] A. De Lucia, R. Oliveto, P. Sgueglia, Incremental approach and user feedbacks: a silver bullet for traceability recovery, in: Proceedings of the International Conference on Software Maintenance, 2006, pp. 299–309.

[28] J. Yang, L. Tan, Inferring semantically related words from software context, in: Proceedings of the Working Conference on Mining Software Repositories, 2012, pp. 161–170.

[29] N. Ali, Y. Gueheneuc, G. Antoniol, Trustrace: Mining software repositories to improve the accuracy of requirement traceability links, IEEE Transaction on Software Engineering 39 (5) (2013) 725–741.

[30] S. K. Bajracharya, J. Ossher, C. V. Lopes, Leveraging usage similarity for effective retrieval of examples in code repositories, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 157–166.

[31] A. Marcus, J. I. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, in: Proceedings of the International Conference on Software Engineering, 2003, pp. 125–135.

[32] M. Kersten, G. C. Murphy, Mylar: a degree-of-interest model for ides, in: Proceedings of the international conference on Aspect-oriented software development, 2005, pp. 159–168.

[33] G. Linden, B. Smith, J. York, Amazon.com recommendations: Item-to-item collaborative filtering, IEEE Internet Computing 7 (1) (2003) 76–80.