An Information Retrieval Approach to Concept Location in Source Code

Andrian Marcus[1], Andrey Sergeyev[1], Václav Rajlich[1], Jonathan I. Maletic[2]

[1]*Department of Computer Science*
Wayne State University
Detroit, Michigan USA 48202
{amarcus,andrey,rajlich}@wayne.edu

[2]*Department of Computer Science*
Kent State University
Kent Ohio 44240
jmaletic@cs.kent.edu

Abstract

Concept location identifies parts of a software system that implement a specific concept that originates from the problem or the solution domain. Concept location is a very common software engineering activity that directly supports software maintenance and evolution tasks such as incremental change and reverse engineering.

This paper addresses the problem of concept location using an advanced information retrieval method, Latent Semantic Indexing (LSI). LSI is used to map concepts expressed in natural language by the programmer to the relevant parts of the source code. Results of a case study on NCSA Mosaic are presented and compared with previously published results of other static methods for concept location.

1 Introduction

The processes of software maintenance and evolution consist of repeated tasks with software change being the most common. Change is triggered by a *change request* that specifies what should be changed in the system. An example of such change request is: "Add a credit card payment to a point-of-sale system". As a part of incremental change design, the programmer must locate the implementation of the concepts embedded in the change request (e.g., "payment", "credit card", etc.). The code implementing these concepts will most likely change.

The users are often the originators of the change requests. While the users of a point-of-sale system could learn about inventory, merchandise, customer, bill payment, and other concepts of the domain, they may know nothing about the implementation of these concepts in the program. The process of *concept location* is to find the code that implements these concepts. In addition to problem domain concepts, change requests can also be formulated in terms of the solution domain (e.g., stack, list, client, server, etc.).

A description of a problem or solution domain concept expressed in natural language is the input to the *concept location process*. The output is a set of software components that implement or address the concept. In the ideal case, the traceability links between external documentation (design and requirements) and the source code provide all that is needed for concept location. In practice though, the external documentation is typically outdated and often nonexistent. An example software system with these types of attributes is an old version of the NCSA Mosaic web browser [24]. We use Version 2.7 of Mosaic in the case study presented in this paper and it has no external documentation. In such situations, the programmer must use other techniques for concept location.

If the programmer fully understands the software, concept location is a relatively easy task. However, it can still be daunting for large complex systems. The difficulty is caused, among other things, by the fact that the input and the output of the location process belong to different levels of abstraction. Namely, the input is at the natural language level and the output is at the source code (i.e., programming language) level. To make the translation from one level to another, extensive knowledge is required. The knowledge includes information on the problem domain, programming techniques, idioms, algorithms, data structures, software architecture, etc.

Concept location is traditionally an intuitive and informal process, based on past experience with the system. When intuition and experience fail to provide the answer, programmers widely use string based pattern matching techniques that take advantage of the similarity of concept names and program identifiers. For example, when searching for the location of "payment", the programmer searches for identifiers such as "payment", "payBill", etc. When the appropriate identifier is found, the programmer studies the surrounding code and decides if this is an actual implementation of the concept, or just a coincidence.

A well-known example of a string pattern matching is the Unix utility `grep`. Although the technique is widely

used, it has known weaknesses. It is based on the correspondence between the name for the concept and an identifier in the code, therefore it fails when the concepts are hidden more implicitly in the code, or when the original programmers used synonyms for the identifiers. Homonyms may also reduce precision of the search.

In this paper we propose a new technique for concept location based on an advanced information retrieval method, namely Latent Semantic Indexing (LSI). The work leverages the experience gained in our previous research where we utilized LSI to support other software engineering tasks that benefit from the power of IR methods. In our earlier work, we used LSI to identify abstract data types and high-level concept clones in procedural code [21, 22]. Additionally, we used LSI to recover traceability links between external documentation and source code [23]. A common feature of our previous work is that LSI was used to find relationships among existing software artifacts (i.e., source to source and source to document).

In the work presented here, we address the concept location problem in the absence of external documentation. This requires a new approach and a different application of LSI than in our previous work. The important difference is that in this application LSI is used to map domain concepts formulated as user queries to software components (i.e., query to source). The users play a dominant part in this process; they formulate the queries and evaluate the results returned by the system.

The users can apply the method in two distinct ways: one is by directly querying the system and the other is based on automatically generated queries. The results for both types of methods for generating queries are evaluated in this paper. Both approaches are significantly different from the way we previously leveraged LSI to support other software engineering tasks. For example, in [23] parts of documentation are automatically mapped to elements of the source code. The user had no input on formulating queries. In [22] the user also did not formulate any queries since the software is automatically clustered. In both cases, the user only inspected and analyzed the results.

Additionally, we compare the results of using our LSI-based method with other known methods of concept location which are based on static code analysis: a search of the program dependency graph and the traditional `grep` based method. By leveraging the strengths of each of these types of approaches in conjunction, we foresee the emergence of very powerful tools to address the problem of static concept location.

The paper is organized as follows. Section 2 covers the related work. Section 3 describes the proposed methodology for concept location. Section 0 presents the results of a case study that is aimed at assessing the quality of the results, and compares LSI based concept location with dependence graph search and the `grep` based search. Section 5 concludes the paper and presents future work.

To make the paper self-contained, a brief conceptual description of LSI is given in the Appendix. This discussion is based on our previous papers and the readers familiar with LSI may skip this part.

2 Related Work

The work presented in this paper addresses two specific issues: the use of information retrieval (IR) methods to support software engineering tasks and activities, and the location of concepts and features in the source code. Related work on concept and feature location is reviewed in subsection 2.1. The overview of the use of IR methods in software engineering is presented in subsection 2.2.

2.1 Concept and Feature Location

The concept assignment problem, as defined by Biggerstaff et al. [3], forms the starting point for much of the work on concept and/or feature location. They describe a research prototype that utilizes parsing, simple clustering, identifier names, and a browser to support concept location.

Chen and Rajlich [4] propose a semi-automated approach for the location of features based on the search of program dependence graph. Other work that addresses the issue of concept or feature location include [12, 18], where reverse engineering techniques and visualization are used to support this problem.

Independently and in parallel, Wilde developed the Software Reconnaissance method [33, 34] which utilizes dynamic information to locate features in existing systems. Wong et al. [35] analyze execution slices of test cases to the same end. Eisenbarth et al. [9] use dynamic information gathered from scenarios of invoking features in a system. This work, based on the analysis of execution traces, is geared towards feature location. Features are special concepts that describe the system functionality, observable at execution. The Software Reconnaissance approach is extended to detect multiple features, represented using concept lattices.

Licata et al [17] uses the user test cases to define feature signature for programs during evolution.

Rajlich and Wilde [25, 26] analyze the importance and role of concepts in program comprehension, describe the process of concept location, and discuss how vital it is to the maintenance of code. Recently, they compared their two approaches in a case study and both approaches proved effective [32]. Each approach seemed better suited for different situations; the dynamic approach is better suited to discover features (i.e., concepts that are observed by the user through the selection of the appropriate input data), while static approaches are better

suited to the location of the remaining concepts that are present in the code but are not selectable by the user at run time.

Other approaches address this problem indirectly or from different perspective. Antoniol et al [1] is looking at identifying the start set during impact analysis. Robillard et al [27, 28] built the FEAT tool for feature separation using concern graphs. Hipikat [7] recommends the user relevant software artifacts when adding new features to Eclipse. Tjortjis [31] uses data mining techniques to help users identify parts of the software related to a concept.

2.2 IR and Software Engineering

Several information retrieval methods exist [29] including signature files, inversion, and clustering. Much of this work deals with indexing, classifying, and retrieving natural language text documents. In software engineering, IR methods are mostly used in the context of indexing reusable software components and automatically constructing libraries [11, 13, 19, 20]. Antoniol et al. [2] use both a probabilistic methods and a Bayesian classifier to address the problem of traceability links between external documentation and source code.

Marcus and Maletic [21, 22] use LSI to derive similarity measures between source code elements. These measures are used to cluster the source code for the identification of abstract data types in procedural code and for the identification of clones. The version of Mosaic that is used here is also used in those case studies. In addition, LSI was used for the recovery of traceability links between external documentation and source code [23]. This approach is compared to that of Antoniol's [2] supporting the conclusion that IR methods can be successfully applied to these types of problems.

3 Using LSI for Concept Location

All techniques for concept location reduce the search space that the user needs to review. However, the user is still necessary in order to locate an actual concept in the code. The same is true for our technique. It is shown in Figure 1 and described in the remainder of this section. Details of LSI are in an Appendix as these details have been published in previous papers. The first part is the preparation of the corpus and generation of the LSI space.

3.1 Preparing the Corpus and the LSI Space

Domain knowledge and concepts are embedded in the source code through identifier names and internal comments. We target these elements from the source code to be analyzed by LSI; therefore a simple preprocessing of the source code is needed. Three actions are taken here: 1) extraction of identifiers and comments; 2) identifier separations; and 3) establishing document granularity.

Extraction of identifiers and comments requires very limited parsing and similarity among the programming languages allows developing a tool that deals with several languages. We developed a simple program that is applicable to C, C++, and Java source code.

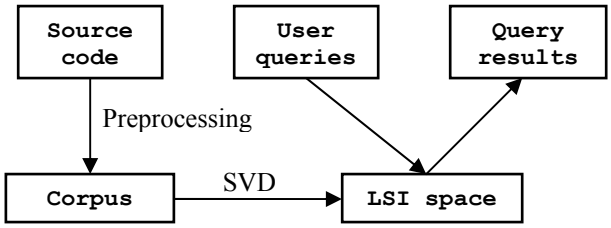

Figure 1: The concept location process using LSI

The next step involves identifier separation. While not paramount with respect to the results, it is a simple step that enriches the corpus and improves the results. We observe two commonly used coding styles for identifiers: one is the combination of words using underscore "_" as separators (e.g., concept_location); and the other is the combination of words using letter capitalization for separation (e.g., ConceptLocation, CONCEPTLocation). All identifiers that follow these rules are separated into constituent words (e.g., concept location for the above examples). The original form of the identifier is also maintained and the separated words are added into the corpus immediately following the identifier and will be later processed by LSI.

The final step of the preprocessing is partition of the code into *documents*. For systems written in procedural languages we choose each function to be a separate document and all declarations blocks outside functions in each file to be treated as one document each.

After the preprocessing, the software system under analysis (S) is decomposed into a set of documents. A *source code document* (or simply document) d is any contiguous set of lines of source code and/or text. Typically a document is a file of source code or a program entity such as a class, function, interface, etc. The *software system* is the complete set of defined documents $S = \{d_1, d_2, ..., d_n\}$.

These steps convert the system into a *corpus*, see Figure 1. Single Value Decomposition (SVD) is then used to create the LSI space. Here SVD is used as a black box subroutine, the definitions and underlying theory can be found in the relevant literature [8, 14, 30]. In the LSI space each document $d_i \in S$ will have a corresponding vector v_i. We use this vector representation to define a similarity measure between two documents $sim(d_i, d_j)$.

The user query will also be converted into a document of LSI space, and the similarity measure between user query and documents of the corpus will help us to identify the documents most relevant to the query.

3.2 Formulating the User Queries

In the proposed methodology, there are two ways in which the user can formulate queries for identifying a given concept. One is to create a natural language query with one or more words, entire phrases, or even short paragraphs. The query is formulated by the users based on the change request and their knowledge of the software domain and/or the source code. The system will return the documents from the software under exploration, ranked by the similarity measure to the query.

The second option for querying is to create queries that contain both words and identifiers from the source code. LSI will automatically provide the similarities between terms. With this in mind, the user can specify a single word query and find all the related terms from the corpus (i.e., identifiers and words from the source code and comments). These queries can be generated automatically starting from a single user specified term or phrase. The user does not need to have a priori knowledge of the terms used in the source code.

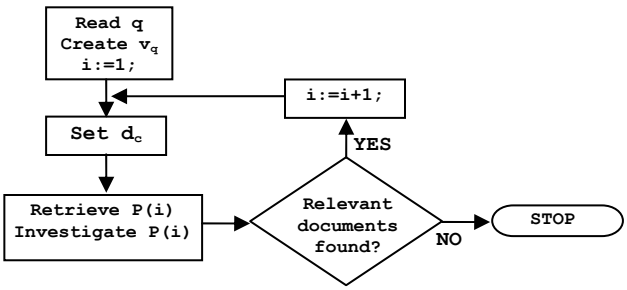

Figure 2: Retrieving the results for a query (q)

In the case study presented in Section 4 we analyzed the results for both types of queries.

3.3 Generating the Results

Using the system is similar with using any search engine. The results are returned as a set of ranked documents that the user inspects to decide their relevancy. A stopping criterion is defined, which indicates the user not to search any further through the returned documents.

Once the user formulates a query, a document q representing the query is created and mapped onto the LSI space. Then LSI creates a corresponding vector v_q and returns the set of all documents in the system, ranked by the similarity measure to the user query. At this point, the user will inspect a subset of the documents in the suggested order and decide which ones are actually parts of the concept.

In order to determine how many documents to inspect, we partition the search space based on the similarity measure. Each partition, at step i, is made up of documents d that are closer than a threshold α to a document d_c: $P(i) = \{d \in S \mid sim(d_c, d) \leq \alpha\}$. The threshold was established empirically. Based on our experience, $\alpha = 0.075$ gives good results.

For the first partition $P(1)$ the user searches, d_c is set to be the closest neighbor of the query q. In a step i, the user will investigate all documents from the partition $P(i)$. If no relevant documents were found, the search stops. If relevant documents were identified, d_c is reset to the last document visited in partition $P(i)$ and the next partition $P(i+1)$ is defined. Formally, the new d_c is reset : $new\ d_c = d_j \in P(i)$ such that $sim(d_c, d_j) = \min\{sim(d_c, d) \mid (\forall)\ d \in P(i)\}$. Figure 2 shows these steps in a flow chart format.

4 Locating "font properties" in Mosaic

The question we try to answer in this case study is how well the LSI-based technique, described in the previous section, helps the user in locating the concepts.

4.1 Case Study Design

Our hypothesis is that LSI-based technique will help the user to identify all documents that correspond to functions and/or declarations that either actually implement the concepts or extensively use it. We want to compare different ways how to formulate the query and the impact of the query on the result. We also want to compare LSI to other previously used or published concept location techniques that are also based on the static analysis of the source code. Techniques that belong to this category include the `grep` based search and the search of the program dependence graph.

It would also be interesting to compare LSI with dynamic techniques but the comparison is less conclusive since the dynamic techniques limit the located concepts to features only, forcing a more narrow selection of the concepts. Since the static and dynamic techniques were compared elsewhere [32], we limited the case study to the comparison among the static techniques.

We conducted a case study to locate concepts in version 2.7 of the NCSA Mosaic web browser [24]. This older version is written in C and we analyzed 269 files with approximately 95,000 lines of comments and code. The choice is motivated by the existence of previous results on concept location in Mosaic [5]. This allows us to compare the results and assess their quality. Mosaic has been used many times in research studies, starting with Clayton and Rugaber [6]. It covers a well-defined domain and is known to the research community.

As in [5], the concept we are locating is "font properties". We want to find the documents within Mosaic that deal with font properties. An example of a change request that may need this concept located is the following: "Add a new font to Mosaic".

In order to minimize the bias, the case study was conducted by the 2nd author of the paper and observed by the remaining authors. However, he was not familiar with

Mosaic and had no previous experience in using LSI. In order to evaluate the results, we used two of the most common measures in experiments with IR methods: *recall* and *precision*. For a given query q, N_i documents will be inspected in step i. Among these N_i documents the user will identify that $C_i \le N_i$ of them are actually related to the concept expressed by the query. There are R_i documents considered relevant to the concept. With these numbers we define the recall and precision for q as follows:

$$\text{Recall} = \frac{C_i}{R_i} = \frac{\#\text{of correct \& retrieved documents}}{\text{total \# of correct documents}}$$

$$\text{Precision} = \frac{C_i}{N_i} = \frac{\#\text{of correct \& retrieved documents}}{\text{total \# of retrieved documents}}$$

If recall is 100%, it means that all the relevant documents are recovered, though there could be recovered documents that are not correct. If the precision is 100%, it means that all the recovered documents are correct, though there could be correct documents that were not recovered.

4.2 Preparation of the Corpus

In this case study, each declaration block, each function, and each ".h file" correspond to a document respectively. Some very large functions and .h files (i.e., over 10,000 characters) were split into smaller pieces. Such documents are later recombined during the evaluation of the results. In other words, the users are unaware that some functions may be mapped to more than one document in the LSI space. This process generated 2,347 documents. Preparation of the corpus is automatic and fast (under 30 seconds for Mosaic), since no serious parsing is needed.

The next phase focuses on the location of a concept in Mosaic (i.e., properties of fonts). In each case we formulated several queries and compared the results.

Table 1: The functions and data types in Mosaic that implement the font properties.

Function/data name	File	Doc
wrapFont()	gui-menubar.c	389
mo_set_fonts()	gui-menubar.c	390
mo_get_font_size_from_res()	gui.c	257
XtResources resources[]	HTML.c	1229
PSfont()	HTML-PSformat.c	1468

4.3 User Specified Queries

We attempted several sets of queries based on the rules described in section 3.2. For the first set of queries we used English terms that we felt best describe the concept. We started with the simplest one: *"font"*. Next, we asked six people in our research lab to formulate a query that describes the font properties, based on their knowledge of web browsers. The results were very similar. Table 2 lists these queries (rows 2 through 7). We also created a query (#8) that is the union of all the words from the other queries. Interestingly, no person used font names in the queries (e.g., Times New Roman, Courier, etc.). When questioned, the consensus was that each of them tried to formulate general (rather than specific) queries in an attempt to reach 100% recall rather than high precision.

After running all these queries, we inspected the source code based on the steps suggested by LSI. The set of relevant documents R, which implement the font properties in Mosaic was determined by the authors. Table 1 shows the functions and data structures that implement the font properties in Mosaic, in what files they are located, and what corresponding document number they have in the LSI space.

Table 2: A first set of queries formulated by different users. The last query (#8) is the union.

#	Query
1	font
2	font size style small regular large
3	font style large small regular family
4	font style bold italics large small regular
5	font size style small regular large family bold italics type
6	font size style small regular large family bold italics
7	font family style bold italics size small regular medium large
8	font size style small regular large family bold italics medium type

Based on the set from Table 1 (five documents), we computed the recall and precision for each of the eight queries in Table 2. Table 3 shows these measures and also shows how many documents we investigated in how many steps. Based on these numbers and the number of relevant documents found in this set, we compute the recall and precision. For some queries (i.e., #1 and #3) some relevant documents were not found before the stopping step was reached (i.e., a step in which no relevant documents are found) which resulted in less than 100% recall. The best result is given by query#4: "*font style bold italics large small regular*". The five documents were located in the four steps, by investigating 15 documents (precision 33.33%).

Table 3: The results for the first set of queries. The highlighted items show the best results.

Q	Investigated documents	Recall	Precision	Last relevant doc. pos.	Step
1	4	0%	0%	89	1
2	54	100%	9.25%	17	4
3	9	60%	33.33%	13	3
4	15	**100%**	**33.33%**	11	4
5	79	100%	6.32%	22	4
6	57	100%	8.77%	14	4
7	49	100%	10.02%	**10**	4
8	72	100%	6.94%	18	4

We also looked on what position is ranked the last relevant document to the query, in other words, how many documents did the user have to visit in order to determine the relevant documents. If we take this as a quality criterion, query#7 did in fact return the best result (i.e., the 5 relevant documents were in the top 10 returned results). The most interesting result though is the fact that query#1 ("*font*") returned the worst results from every point of view. In fact, the first four returned documents (returned in step 1) are not part of our concept. The first relevant document is in fact on the 5th position. This did not come as a surprise since the term *font* is related to a number of other concepts such as "font stack".

Except query#1, each query returned the correct results within the first 22 documents (see Table 3). Even for query#3, where we stopped after investigating the first 9 documents, all the relevant documents were in the top 13 returned documents.

4.4 Automatically Generating Queries

The second part of the case study is aimed at establishing how well the system can automatically generate queries, starting from a single word or phrase specified by the user.

A second set of queries are defined using LSI starting from the word "font". We use the top 40 terms from the Mosaic corpus that are most related to "font". These are, as returned by LSI: *nheader, medium, naddress, nfixeditalic, nplain, nplainbold, nplainitalic, nsup, wb, nbold, nfixed, nfixedbold, nitalic, normal, nfont, nactive, wrap, super, subfield, lsa, times, sophisticated, helvetica, family, plainbold, fixeditalic, plainitalic, century, fixedbold, schoolbook, xmx, set, lucidatypewriter, nlisting, nresolve, previously, lucidabright, wbc, subscript, superscript*. With these terms we automatically created 40 queries based on the following formula:

```
query(n) = "font" + the first n terms
           from the list
```

Based on the previous queries we learned that the term "font" returns rather poor results since is widely used in Mosaic and generates many correlations. Therefore, we generated another set of 40 queries similar to the ones above, but without the term "font" included:

```
query'(n) = the first n terms from the list
```

For this set of queries, we looked within how many positions in the ordered list of returned documents are the five relevant ones, the same information as "Last relevant document position" in Table 3. The results of the comparison between the two sets of queries are shown in Figure 3. The best result (35) among the first set of queries is given by query(31) (it contains the word "font" followed by the 31 closest terms). Among the latter set the best result (i.e., 11) is given by both query'(30) and query'(31) (contain the first 30 and 31 respectively closest terms to "font"). On average, in the first set the relevant results were among the first 90 returned documents. In the second set the results are among the first 77 returned documents, on average.

We also computed the precision and recall for the queries in these two sets, considering all documents returned by LSI in each step and the defined stopping criterion. For *query(31)* we investigated 15 documents in 5 steps and identified four of the five relevant documents: recall 80% and precision 26.66%. For *query'(31)* we investigated 64 documents in 6 steps and found all five relevant results: recall 100% and precision 7.80%. For this query the last relevant document (389) is in fact the nearest neighbor (d_c) in the last examined partition.

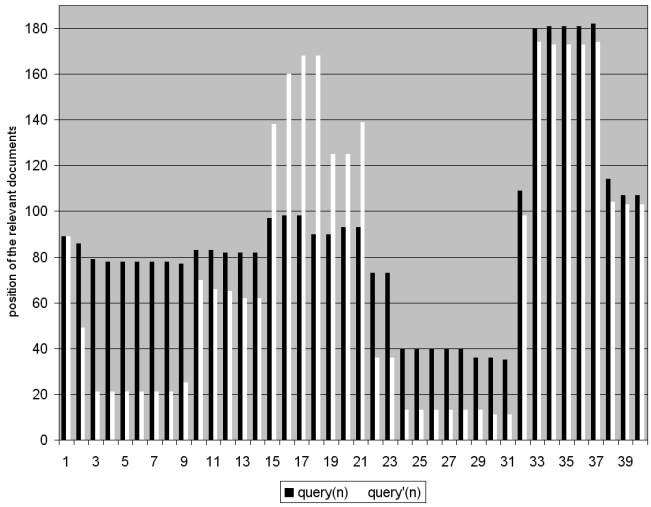

Figure 3: Position of the 5 relevant documents for each query(n) and query'(n), with n=1, …, 40.

In the worst case there is *query(37)* where 40 documents are examined in 5 steps with 80% recall and 10% precision, and *query'(37)* where five documents are investigated in 2 steps with 20% recall and 10% precision.

While the average results are not as good as in the case of the user formulated queries, the best result is almost the same. One important thing to remember is that these 80 queries are in fact automatically generated starting with the word "font". No domain knowledge is needed to formulate the queries, as in the first set.

4.5 Comparison with Search of the Dependence Graph

These results were also compared with those published in [5], which are based on searching the dependence graph. Some discrepancies occurred; namely only three of the five detected component in our process correspond to those detected previously in [5] (i.e., *mo_set_fonts()*, *mo_get_font_size_from_res()*, and *XtResource resources[]*). The authors in [5] also identified one more function and three global variables (i.e., *menubar_cb()*, *Rdata*, *menuspec*, and *mo_token*). At the same time they

did not identify two of the functions we identified (i.e., *PSFont()* and *wrapFont()*).

In order to understand and explain the difference we must emphasize the differences in the approaches and underlying maintenance tasks. In [5] the task under analysis by the authors was a change request to add a new font size *Tiny*. In order to locate the code where the change needs to be done, the authors divided the process in four subtasks, based on prior knowledge of the system: 1) to find the function that opens a new window; 2) to find how the font properties are specified in the new window; 3) to find what the default type is, and how and where it is set; and 4) to find the connection between the font - related menu items and the font settings. The end result in [5] is the impact set of the change. In contrast, our goal is to identify the parts of the software that implement font properties, rather than the impact set.

With this in mind and after the investigation of the source code the reasons for the discrepancies became clear. Our approach missed the *Rdata*, *menuspec*, and *mo_token* global variables because of the selected granularity size. The missed variables are defined inside declaration blocks with many other global variables that did not relate to font properties. In addition, they support menu handling and only one of the multiple menus actually dealt with selections of font types. Thus, the correlation with font related terms is very weak. The same reason stands for the *menubar_cb()* function. We consider this function simply a usage of the font property concept rather than a definition. These documents should be included in the impact set as they change in response to the change request.

Based on the same principle, the authors in [5] did not include *wrapFont()* in their impact set, although they did visit it. While, *wrapFont()* is part of the concept definition (i.e., it is called only by *mo_set_fonts()* and does not call any function) it did not need to be changed for the introduction of a new font type.

Finally, *PSFont()* was not identified in the case study presented in [5] for two reasons. The authors started the concept location from the "open new window" menu and identified the font properties related functions through data and control dependencies. With respect to this feature (i.e., display of fonts in windows) their impact set is correct and the change is propagated through all the necessary functions. However, *PSFont()* does set the font properties for post script printing of the HTML page that is not related to window opening and display functionality. Based on the approach from [5] in order to identify *PSFont()* the logical starting point would have been in the "print post script" menu. In addition, there is no explicit dependency between the *PSFont()* and the *menubar_cb()* function, which was the first function identified in the impact set.

This finding turned out to be an unexpected result and helped in better understanding the differences between methods. In essence, the "font properties" concept is part of at least two features in Mosaic: "display in window" and "print postscript". In conclusion, we are able to find a part of the concept missed in the previous case studies. In fact, the document corresponding to *PSFont()* is the nearest neighbor to each query in our case study, except query#1. Although direct comparison of recall and precision is questionable given that we identified slightly different sets of functions and data, it is an indication of a qualitative aspect of the result. In [5] the precision for the entire location process was 7.69% and for the propagation process was 5.63%. Precision using LSI (Table 3) is close to this result in the worst case, but better in the best case.

Finally, for each of the queries, there are a number of documents not relevant to the searched concept, which always occurres among the top 10-15 nearest neighbors to the query. They are: *PopFont(), PushFont(), TriggerMarkChanges(), HTMLPart,* and *font_rec*. The first two are obviously related by usage to the font properties as both concepts (i.e., font property and font stack) are part of the more general concept of font. The *TriggerMarkChanges()* function is a very large function in Mosaic that deals with changes and history, as well as *font_rec*. The function uses the font stack. *HMTLPart* is a large structure that defines the properties for new HTML widgets, including the displayable fonts. Finally, one large block of macro definition from *HTML.c* is of interest since it defines names for new resources including some related to fonts. Based on the task at hand for which the concept location is needed, these documents may or may not be considered relevant by the user.

4.6 Comparison with grep-type Search

As mentioned previously, one of the most commonly used methods in practice for concept location is the `grep` based search. Similar comparisons were made by Maarek et al. [19] and Antoniol et al. [2]. Given this fact we used the regular expression search engine built in Microsoft Visual .NET development environment to search the Mosaic source code. It is very similar in usage with `grep`. In fact, it has some additional features. One of the issues with the `grep` based approach is also the granularity level they work at (i.e., files). This prohibits us to directly compare recall and precision, since the results are in different format.

The obvious start is to look for the word "font", which returned 1168 occurrences in 24 files, an obviously ineffective result. We then performed several searches by combining words with the "*or"* operator. These still proved to be unsuccessful since they still returned hundreds of hits in dozens of files. Then we created queries using regular expressions. The best results were returned by:

1. *font[^]*style*
2. *font[^]*large*.

For query (1) we obtained 9 occurrences in 1 file that pointed to the *PSFont()* function. For query (2) we obtained 9 occurrences over three files that pointed to *PSFont()* and *mo_get_font_size_from_res()*. These results helped locating at least a part of the implementation.

Other queries such as:

- *font[^]*properties*
- *[^]*font[^]*size[^]*style[^]*small[^]*regular[^]*large[^]**

returned no hits. In conclusion, a number of the queries returned results that were not very helpful. Even the better ones missed many of the relevant documents, hence recall was low. The main problem with the `grep` based approach is that the returned results are not ranked. This means that the user has to examine a large number of documents with the same priority.

5 Conclusions and Future Work

The paper presents a new technique for concept location using an information retrieval method, latent semantic indexing (LSI). The method uses LSI to find semantic similarities between user queries and modules of the software in order to locate concepts of interest in the source code. Two variants of the concept location technique using LSI are presented. One, based on user formulated queries and the other based on partially automated generated queries.

A case study of locating concepts in NCSA Mosaic is also presented and analyzed. The results are compared with other methods that are based on regular expression searches and search on the program dependence graph.

By comparison with related methods, the use of LSI for concept location presents several advantages. The method is almost as easy and flexible to use as `grep` based techniques and it provides better results. Additionally, we are able to identify certain parts of a concept (i.e., the *PSFont()* function) that are missed by the dependence graph search approach. The advantage of using LSI is that the method is independent of programming language, and the source code preprocessing is simpler than building a dependence graph.

One important feature of the method that sets it apart from other related approaches is that LSI is able to identify words and identifiers from the source code that are related to a user-specified term or phrase within the context of the software system. This allows us to automatically generate queries starting with a single (or more) user-specified word. These queries returned results comparable with the queries formulated manually by the users based on their domain knowledge.

Several additional issues will be addressed in future work. As far as the quality of the corpus is concerned, we plan a set of case studies with software that includes external documentation, and software that is commented more richly than Mosaic. We will investigate the impact of these additional properties on the results. In addition, we plan to see how much the structure of the software (e.g., procedural vs. object-oriented, application vs. library) influences the results.

With respect to the user queries we plan to define several query templates based on the type of concept that is searched. One important aspect that needs to be addressed is the definition of better heuristics, dependent on the corpus that will allow a flexible definition of the α threshold to determine the stopping criterion. This should improve the precision of the method. We also plan to define a heuristic that will determine which of the automatically generated queries is best. In the same realm, we plan to formulate a third method for user queries. It will automatically translate the user queries from natural language into terms from the software system vocabulary, based on a modified editing distance between the terms. Future case studies will assess these types of queries as well.

The results of this paper and the conclusions in [32] show that none of the concept location methods is perfect. The logical conclusion is that the user should use a combination of such methods when searching for concepts in the source code. An investigation on how to combine various methods to support location of concepts during maintenance activities is being undertaken.

6 Acknowledgements

This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

7 References

[1] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: a Case Study", in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'00), Zurich, Switzerland, February 29 - March 3 2000, pp. 227-230.

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", IEEE Transactions on Software Engineering, vol. 28, no. 10, October 2002, pp. 970 - 983.

[3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem", Comm. of the ACM, vol. 37(5), May 1994, pp. 72-82.

[4] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependency Graph", in Proceedings of Intern. Workshop on Program Comprehension (IWPC'00), 2000, pp. 241-249.

[5] Chen, K. and Rajlich, V., "RIPPLES: Tool for Change in Legacy Software", in Proceedings of International Conference on Software Maintenance (ICSM'01), 2001, pp. 230 - 239.

[6] Clayton, R., Rugaber, S., Taylor, L., and Wills, L., "A Case Study of Domain-based Program Understanding", in Proceedings of 5th Workshop on Program Comprehension, Dearborn, MI, May 28-30 1997, pp. 102-110.

[7] Cubranic, D. and Murphy, G. C., "Hipikat: Recommending pertinent software development artifacts", in Proceedings of 25th International Conference on Software Engineering (ICSE'03), Portland, OR, May 2003, pp. 408-418.

[8] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", J. of the Amer. Soc. for Info Science, vol. 41, 1990, pp. 391-407.

[9] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", IEEE Transactions on Software Engineering, vol. 29, no. 3, March 2003, pp. 210 - 224.

[10] Etzkorn, L. H. and Davis, C. G., "Automatically Identifying Reusable OO Legacy Code", IEEE Computer, vol. 30, no. 10, October 1997, pp. 66-72.

[11] Fischer, B., "Specification-Based Browsing of Software Component Libraries", in Proceedings of ASE, 1998, pp. 74-83.

[12] Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E., "A Cliche'-Based Environment to Support Architectural Reverse Engineering", in Proceedings of Intern Conference on Software Maintenance (ICSM '96), Nov 04 - 08 1996, pp. 319-328.

[13] Frakes, W., "Software Reuse Through Information Retrieval", in Proc of HICSS, Kona, HI, Jan. 1987, pp. 530-535.

[14] Landauer, T. K., Foltz, P. W., and Laham, D., "An Introduction to Latent Semantic Analysis", Discourse Processes, vol. 25, no. 2&3, 1998, pp. 259-284.

[15] Landauer, T. K., Laham, D., and Foltz, P. W., "Learning human-like knowledge by Singular Value Decomposition: A progress report", Advances in Neural Information Processing Systems, vol. 10, 1998, pp. 45-51.

[16] Landauer, T. K., Laham, D., Rehder, B., and Shreiner, M. E., "How Well Can Passage meaning Be Derived without Using Word Order? A Comparison of Latent Semantic Analysis and Humans", in Proceedings of Annual Conference of the Cognitive Science Society, 1997, pp. 412--417.

[17] Licata, D. R., Harris, C. D., and Krishnamurthi, S., "The feature signatures of evolving programs", in Proceedings of IEEE International Conference on Automated Software Engineering (ASE'03), October 6-10 2003, pp. 281-285.

[18] Lukoit, K., Wilde, N., Stowell, S., and Hennessey, T., "TraceGraph: Immediate Visual Location of Software Features", in Proceedings of International Conference on Software Maintenance (ICSM'00), San Jose, Oct 11 - 14 2000, pp. 33-39.

[19] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", IEEE Transactions on Software Engineering, vol. 17, no. 8, 1991, pp. 800-813.

[20] Maarek, Y. S. and Smadja, F. A., "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proceedings of SIGIR, Cambridge, June 1989, pp. 198-206.

[21] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings of Intern. Conference on Software Engineering (ICSE'01), Toronto, Canada, May 12-19 2001, pp. 103-112.

[22] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[23] Marcus, A. and Maletic, J. I., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in Proc Intern Conference on Software Engineering (ICSE'03), Portland, OR, May 3-10 2003, pp. 125-137.

[24] Mosaic, "Mosaic Source Code v2.7b5", NCSA, ftp site, Date Accessed: 4/12/2000, ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/.

[25] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in Proceedings of International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 27 - 29 2002, pp. 271-278.

[26] Rajlich, V., Wilde, N., Buckellew, M., and Page, H., "Software Cultures and Evolution", IEEE Computer, vol. 34, no. 9, September 2001, pp. 24-28.

[27] Robillard, M. P. and Murphy, G. C., "Concern graphs: finding and describing concerns using structural program dependencies", in Proceedings of Intern Conference on Software Engineering (ICSE 2002), Orlando, 2002, pp. 406 - 416.

[28] Robillard, M. P. and Murphy, G. C., "Automatically Inferring Concern Code from Program Investigation Activities", in Proceedings of 18th International Conference on Automated Software Engineering (ASE'03), October 2003, pp. 225-234.

[29] Salton, G., Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer, Addison-Wesley, 1989.

[30] Salton, G. and McGill, M., Introduction to Modern Information Retrival, McGraw-Hill, 1983.

[31] Tjortjis, C., Sinos, L., and Layzell, P. J., "Facilitating Program Comprehension by Mining Association Rules from Source Code", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, May 10-11 2003, pp. 125-133.

[32] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., "A Comparison of Methods for Locating Features in Legacy Software", Journal of Systems and Software, vol. 65, no. 2, 15 February 2003 2003, pp. 105-114.

[33] Wilde, N. and Gust, T., "Locating User Functionality in Old Code", in Proceedings of Conference on Software Maintenance, Orlando, Florida, 1992, pp. 200-205.

[34] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", Journal of Software Maintenance and Evolution, vol. 7, 1995, pp. 49-62.

[35] Wong, E., Gokhale, W., S., S., and Horgan, J. R., "Quantifying the closeness between program components and features", Journal of Systems and Software, vol. 54, no. 2, Oct. 2000, pp. 87-98.

Appendix: Overview of LSI

Latent Semantic Indexing (LSI) [14] is a machine-learning model that induces representations of the meaning of words by analyzing the relation between words and passages in large bodies of text. LSI has been used in applied settings with a high degree of success in areas like automatic essay grading and automatic tutoring to improve summarization skills in children. As a model, LSI's most impressive achievements have been in human language acquisition simulations and in modeling of high-

level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity. For complete details on LSI see [8]. LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with vector space model (VSM) [30] approaches. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix decomposed using Singular Value Decomposition (see below). It has been shown in [14, 15] that LSI addresses the synonyms very well. With simple corpus training, LSI managed to answer correctly 64% of the synonyms questions in the Test of English as a Foreign Language, better than the average student.

VSM is a widely used classic method for constructing vector representations for documents. It encodes a document collection by a term-by-document co-occurrence matrix whose [i, j]th element indicates the association between the ith term and jth document. In typical applications of VSM, a term is a word, and a document is an article. However, it is possible to use different types of text units. For instance, phrases or word/character n-grams can be used as terms, and documents can be paragraphs, sequences of n consecutive characters, or sentences. The essence of VSM is that it represents one type of text unit (documents) by its association with the other type of text unit (terms) where the association is measured by explicit evidence based on term occurrences in the documents. A geometric view of a term-by-document matrix is as a set of document vectors occupying a vector space spanned by terms; we call this vector space VSM space. The similarity between documents is typically measured by the cosine between the corresponding vectors, which increases as more terms are shared. In general, two documents are considered similar if their corresponding vectors in the VSM space point in the same (general) direction.

LSI relies on a Single Value Decomposition (SVD) [30] of the co-occurrence matrix. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a feature space without serious loss of specificity. The formalism behind SVD is rather complex and lengthy to be presented here. The interested reader is referred to [30] for details. One of the most successful applications of SVD in information retrieval is the Google search engine (www.google.com).

Any matrix can be decomposed and then recomposed perfectly using only as many factors as the smallest dimension of the original matrix. However, an interesting phenomenon occurs when the original matrix is recomposed using fewer dimensions than necessary: the reconstructed matrix is a least-squares best fit.

Intuitively, in SVD a rectangular matrix X is decomposed into the product of three other matrices. One component matrix (U) describes the original row entities as vectors of derived orthogonal factor values, another (V) describes the original column entities in the same way, and the third is a diagonal matrix (Σ) containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed (i.e., $X = U\Sigma V^T$). The columns of U and V are the left and right singular vectors, respectively, corresponding to the monotonically decreasing (in value) diagonal elements of Σ which are called the singular values of the matrix X. When fewer than the necessary number of factors are used, the reconstructed matrix is a least-squares best fit. One can reduce the dimensionality of the solution simply by deleting coefficients in the diagonal matrix, ordinarily starting with the smallest. The first k columns of the U and V matrices and the first (largest) k singular values of X are used to construct a rank-k approximation to X through $X_k = U_k\Sigma_k V_k^T$. The columns of U and V are orthogonal, such that $U^TU = V^TV = I_r$, where r is the rank of the matrix X. X_k constructed from the k-largest singular triplets of X (a singular value and its corresponding left and right singular vectors are referred to as a singular triplet), is the closest rank-k approximation (in the least squares sense) to X.

With regard to LSI, X_k is the closest k-dimensional approximation to the original term-document space represented by the incidence matrix X.

For document retrieval in the LSI space a similarity measure is defined between two documents as the cosine between their corresponding vectors in the LSI space. The similarity measure between two documents d_q and d_i is defined as a cosine $sim(d_q, d_i) = cos(v_q, v_i)$. We denote the inner product of the two vectors v_q and v_i as $v_q^T v_i$ and length of a vector v as $|v|$. The *cosine* of v_q and v_i is the length-normalized inner product:

$$\cos(v_q, v_i) = \frac{v_q^T v_i}{|v_q|_2 \times |v_i|_2}$$

LSI is mostly used on natural language corpora. However, the method lends itself perfectly to other type of data. One criticism of this type of method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. Very good representations and results are derived without this information [16]. This characteristic is well suited to the domain of source code and internal documentation. Source code is hardly English prose but with selective naming, much of the high level meaning of the problem-at-hand is conveyed to the reader. Internal source code documentation is also commonly written in a subset of English [10] so queries formulated in natural language are perfectly usable. This makes automation drastically easier and directly supports programmer defined variable names that have implied meanings (e.g., avg) yet are not in the natural language vocabulary.