Improving Feature Location by Enhancing Source Code with Stereotypes

Nouh Alhindawi[1], Natalia Dragan[1], Michael L. Collard[2], Jonathan I. Maletic[1]

[1]Department of Computer Science
Kent State University
Kent, Ohio, USA
{nalhinda, ndragan, jmaletic}@kent.edu

[2]Department of Computer Science
University of Akron
Akron, Ohio, USA
collard@uakron.edu

***Abstract*—A novel approach to improve feature location by enhancing the corpus (i.e., source code) with static information is presented. An information retrieval method, namely Latent Semantic Indexing (LSI), is used for feature location. Adding stereotype information to each method/function enhances the corpus. Stereotypes are terms that describe the abstract role of a method, for example get, set, and predicate are well-known method stereotypes. Each method in the system is automatically stereotyped via a static-analysis approach. Experimental comparisons of using LSI for feature location with, and without, stereotype information are conducted on a set of open-source systems. The results show that the added information improves the recall and precision in the context of feature location. Moreover, the use of stereotype information decreases the total effort that a developer would need to expend to locate relevant methods of the feature.**

***Keywords*: software maintenance, information retrieval, feature location, method stereotypes, program comprehension.**

I. Introduction

When correcting a fault, adding a new feature, or adapting a system to conform to a new platform or API, software engineers must first find the relevant parts of the code that correspond to the particular change. This is termed feature or concept location [1, 2]. Feature location involves searching, exploring, reading, and understanding the source code. These types of comprehension activities make up a major portion of the costs in the evolution of modern software systems [3, 4].

A number of different techniques to support feature location have been suggested and involve everything from simple regular-expression matching, dynamic and static program analysis, and information-retrieval techniques. Regular-expression matching is often used by programmers but returns far too many false positives and has no ability to rank the results. Static and dynamic methods often suffer from the same types of problems [2, 5] (too many false positives) or require very accurate test cases for the feature, which may not be available.

Over the past few years' information-retrieval methods have been used for feature location with encouraging results [2, 6-12]. Information retrieval (IR) methods move far beyond keyword matching and regular expressions and use advanced probability and information theory to derive relationships between documents based on the vocabulary and occurrences of words in each document. This is attractive because queries (to find particular) features can be made in the language of the documents (i.e., programming language terms, identifiers, and natural language of comments). There are also means to rank the results from a query, much like the manner that web search engines present results.

While the use of IR methods has been successful for feature location, there is room for improvement. In particular, false positives are an issue and the most relevant documents are not always ranked highly. This presents problems for software engineers using tools for feature location. Adoption is a problem because results are not good enough and searching through a long list of possible relevant documents is costly and time consuming.

To address this problem a number of researchers have combined various static and dynamic analysis techniques with the results from IR methods. For example, Formal Concept Analysis (FCA) has been used to help rank the results produced by IR methods [10]. The approach taken here addresses the problem of improving the results of IR methods in a novel and very different manner.

We use IR methods in a standard manner [8, 10, 13] for the problem of feature location. However, before applying the IR method, we enhance the corpus (i.e., source code) with new information. This new information is derived automatically from the source code via static program analysis. Specifically, we re-document the source code (i.e., add new terms) with stereotype [14] information for each method/function in the system. After this has been completed we use the IR method to run queries for feature location. This type of up-front enhancement of a corpus to improve results has not been investigated previously.

Adding these new terms is a form of supervision added on top of an unsupervised method (i.e., LSI). Apriori knowledge is often used to direct and supervise machine-learning and information-retrieval approaches [15]. Here, we derive this information from the corpus itself. Others have used similar approaches based on ontological information [16] and inferred semantics from term distribution [17]. From an information theoretic standpoint the addition of (relevant) information will improve the results of an information-retrieval technique [18, 19]. That is, more information is better, so long as you don't add noise.

Our hypothesis is that the stereotype information is relevant and will improve the results in the context of feature location. Our experimental study supports this hypothesis. The results demonstrate a definite improvement in locating relevant methods pertaining to the feature being queried when stereotype information is included.

We chose to use stereotype information for a number of reasons. Stereotypes describe the abstract behavior and role of a method within a class. We felt that this was relevant

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.41

information and our previous work investigating the automatic detection of stereotypes [14, 20], bares evidence that they support program comprehension. Moreover, we found that distributions of method stereotypes can be used to derive class stereotypes. This evidence gave support to enhancing the information within the source code. Lastly, stereotype information is new information that did not previously exist in the source code (i.e., new vocabulary).

The remainder of this paper is organized as follows. First in Section II we present related work on using information retrieval methods for feature location. Section III presents a taxonomy of method stereotypes and a brief description of how they are computed and added to the source code. The approach of using LSI and stereotypes together is described in Section IV. An experiment comparing the results of using LSI with the additional stereotype information is given in Section V. This is followed by discussion of the results in Section VI and threats to validity in Section VII.

II. Related Work

An overview of existing static feature location approaches is reviewed along with related work on feature location using LSI.

A. Previous Work on Feature Location

Historically, developers used pattern-matching techniques like *grep* to locate features in the source code. Using pattern-matching techniques is simple; it performs an investigation through pattern matching on character strings. Nevertheless, it requires a lot of experience from the developer. If the technique failed, more advanced tools were required, especially when the system is large [2, 4, 6, 7, 21].

Biggerstaff et al. [1] referred to concept location as the concept location assignment problem. Their work was a preliminary point for many efforts to facilitate and develop the process of concept location. Call graphs, program clustering graphs, etc. are used in their approach.

Chen and Rajlich [22] presented an approach based on using an Abstract System Dependencies Graph (ASDG). The ASDG can lead, guide, and assist the user in the process of searching for a particular feature.

Wilde [23] developed the software-reconnaissance method, which utilizes dynamic information to locate features in existing systems. Wong et al. [24] analyzed the execution slices of test cases to the same end. Eisenbarth et al. [5] used dynamic information gathered from scenarios of invoking features in a system to locate the features in source code. Tools that deal with feature location are either static or dynamic. Overlap between features cannot be distinguished using dynamic analysis, while static analyses do not often identify units contributing to a particular execution scenario [2, 4].

Revelle and Poshyvanyk [25] presented an investigative study of ten feature-location techniques that use different combinations of textual, dynamic, and static analyses. A survey of feature location techniques is presented in [2].

B. Previous Work on Feature Location using IR

Recently, IR methods have been used successfully and effectively for feature location [2, 6-12]. For more details, we refer the readers to the survey by Binkley and Lawrie [4] on information-retrieval applications in software maintenance and evolution.

Marcus and Maletic [11] were the first researchers to use LSI for application to software engineering. They obtained similarity measures between source-code components in order to cluster and classify these components. They also defined a number of comprehension metrics. These metrics use the profile produced by the application of LSI to a matrix of source code. In [6], Marcus et al. linked LSI to concept location, where they used LSI to map concepts expressed in change requests described using natural language to the relevant source-code components.

Many efforts have been made to improve the use of LSI in feature location by adding or integrating meaningful information to the whole process of feature location [8, 10]. For example, in [8], the authors combined LSI with user-execution scenarios to improve the accuracy of feature location.

Poshyvanyk et al. [13] proposed a Visual Studio plugin called IRiSS which, based on the existing "find" feature, uses LSI to search projects using natural-language queries. In [9], Poshyvanyk et al. combined static and dynamic techniques they had previously developed. Their goal was to use both certain and uncertain knowledge extracted with both static and dynamic analyses, filter it by probabilistic and information-retrieval techniques, and in this way to identify features in source code. Moreover, Poshyvanyk et al. [7], in order to improve the accuracy of feature location process, proposed a technique that combines information from an execution trace and from the comments and identifiers in the source code. M. Revelle et al. [12] applied advanced web mining algorithms (*Hyperlinked-Induced Topic Search* (HITS) and *PageRank*) to analyze execution information during feature location. Their approach improved the effectiveness of existing approaches by as much as 62%. The ability of LSI in providing a straightforward language-independent method that recognizes relationships between documents is shown in SNIAFL [26].

The dimensions of singular value decomposition when using LSI have been studied. The range of 200 to 300 dimensions has been proposed as a "golden standard" [6]. In [9], Poshyvanyk et al. looked at varying the number of dimensions when using LSI. Their findings concluded that any larger factor could improve the results but would generate too large a search space.

Generally, the current approaches either use IR methods alone or in combination with other techniques, such as [7, 9]. There is a need for improvement in recall and precision of feature location. None of these approaches augment the source code with new information. In our approach, the source code is augmented with method stereotypes, which is described next.

III. Method Stereotypes

Stereotypes are a concise abstraction of a method's role and responsibility in a class and system [14]. They are widely used

to informally describe methods. Stereotypes for classes are also used in the same manner to describe their role and responsibility within a system's design. UML provides mechanisms for documenting class stereotypes. Manually documenting method stereotypes is relatively easy for a small number of classes and methods however it is quite costly to do so for an entire system.

A taxonomy of method stereotypes (see Table 1) and technique to automatically reverse engineer stereotypes for existing methods was presented by Dragan et al. in [14]. This work was further extended to support the automatic identification of class stereotypes in [20]. That work describes an approach to automatically identify method stereotypes that we use in this research. We refer the readers to those works for complete details on computing method stereotypes; however we present the main points here.

TABLE 1 TAXONOMY OF METHOD STEREOTYPES AS GIVEN IN [20]. THE TAXONOMY IS MAINLY FOCUSED ON THE C++ PROGRAMMING LANGAUGE. METHODS MAY BE LABELED WITH ONE OR MORE STEREOTYPES.

Stereotype Category	Stereotype	Description
Structural ***Accessor***	get	Returns a data member.
	predicate	Returns Boolean value which is not a data member.
	property	Returns information about data members.
	void-accessor	Returns information through a parameter.
Structural ***Mutator***	set	Sets a data member.
	command	Performs a complex change to the object's state *(this)*.
	non-void-command	
Creational	constructor, copy-const, destructor, factory	Creates and/or destroys objects.
Collaborational	collaborator	Works with objects (parameter, local variable and return object).
	controller	Changes an external object's state (not *this*).
Degenerate	incidental	Does not read/change the object's state.
	empty	Has no statements.

The taxonomy of method stereotypes given in Table 1 unifies and extends previous literature on stereotypes and addresses a number of gaps and deficiencies that were present. The taxonomy was developed primarily for C++ but many aspects of it can be applied to other programming languages. Based on this taxonomy, static program analysis is used to determine the stereotype for each method in an existing system. The taxonomy is organized by the main role of a method while emphasizing the creational, structural, and collaborational aspects with respect to a class's design. Structural methods provide and support the structure of the class. For example, accessors read an object's state, while mutators change it. Creational methods create or destroy objects of the class. Collaborational methods characterize the communication between objects and how objects are controlled in the system. Degenerate are methods where the structural or collaborational stereotypes are limited. The naming is based on the mathematical term for a case for which a stereotype cannot be any simpler. Also, a method may have more than one stereotype. A tool [14], *StereoCode*, was developed that analyzes and re-documents C++ source code with the stereotype information for each method. Re-documenting the source code is based on srcML (Source Code Markup Language) [27], an XML representation of source code that supports easy static analysis of the code.

In order to provide the method-stereotype identification, we translate the source code into srcML, and then, *StereoCode* takes over by leveraging XPath, an XML standard for addressing locations in XML. For details about the rules for identifying each method stereotype, we refer the readers to [14]. Adding the comments (annotations) to source code is quite efficient in the context of srcML. The XPath query gives us a location of the method and we can then do a simple transformation within the srcML document to add the necessary comments. This process is fully automated and very efficient/scalable. Running *StereoCode* on two systems used in the evaluation takes less than a minute each.

```
class  DataSource :public Observable
{
…
public:
  /** @stereotype get */
  const string& getName() const;

  /** @stereotype predicate */
  bool isValidLabel(const string& label) const;

  /** @stereotype command */
  virtual void reserve(int count );
…
};
```

Figure 1. A code snippit of the HippoDraw C++ Class DataSource after re-documenting with the method stereotypes.

Methods can be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have secondary stereotypes from the collaborational and degenerate categories. For example, a two-stereotype method *get-collaborator* returns a data member that is an object or uses an object as a parameter or a local variable.

Figure 1 presents an example of stereotype labeling for part of the class DataSource from the HippoDraw open-source application (one of the systems used in the experiment). The class DataSource supplies one or more arrays of data.

The evaluation of the taxonomy and approach demonstrated two things. First, the method-stereotype taxonomy covered a very large percentage of the methods studied. That is, almost all methods can be labeled by the classification scheme. Second, the tool re-documents systems according to the taxonomy with very high accuracy in comparison to human evaluation.

IV. LSI+STEREOTYPES FOR FEATURE LOCATION

We now describe the approach taken for feature location. The same approach as taken in [6] is used here. The IR method, Latent Semantic Indexing (LSI) [4, 28] is the basis of the approach. We term our approach LSI+S (LSI plus stereotypes) to differentiate it using with LSI without stereotypes.

We start with the source code for a software system. As described in the previous section, our *StereoCode* tool is applied to automatically determine the stereotype of each method and re-document it with a comment stating its stereotype. Next preprocessing is done to the resultant re-documented source code to convert it into input for LSI. This is termed a corpus (we will describe how the corpus is generated below in section B).

At this point LSI is applied to the corpus. A co-occurrence matrix of vocabulary × documents is computed and Singular Valued Decomposition (SVD) [29] is applied to reduce the dimensionality of this matrix by exploiting the co-occurrence of related terms. The result is a subspace that can be queried against to locate documents most similar to the query phrase. Ranked documents will be retrieved based on their similarities to the query. The user then inspects the results. More details about these steps are covered separately on the following sub-sections. We now give a brief overview of LSI and describe the details of how we set up the feature location process.

A. Latent Semantic Indexing

LSI is a corpus-based statistical technique which is used for inducing and representing characteristics of the meanings of words and passages (of natural language) reflective in their usage [6, 28].

Among code-based feature-location techniques, LSI is considered one of the better techniques capable of recognizing terms in source code that are relevant to a user query [4]. Moreover, LSI is language independent and using it to preprocess and query the source code is more efficient than using a pattern-matching technique, especially with its capability in dealing with synonymy and polysemy. It is also simpler than using graph-based techniques [4, 6].

The initial step of the IR process is to build the corpus for the software system. The corpus consists of a set of documents. In this work (in most all feature location works), documents in the corpus are methods or functions. These documents include the text of each method including all the identifier names, comments, etc.

B. Corpus Creation

Constructing the corpus is an important step for feature location using LSI. Five actions are taken to create the corpus: 1) Extraction of identifiers, and comments; 2) Extraction of method stereotypes; 3) Identifier (term) separations; 4) Removing stop words; 5) Division into documents (method level).

A well-built corpus helps in locating the relevant methods (effectiveness measure). As mentioned in [12], not all feature-location techniques can locate all feature-relevant methods, and one of the reasons behind that is the preprocessing steps taken by each technique when building or creating the corpus. We developed an efficient corpus builder in C++ to extract these important elements from source code using srcML [27]. It takes less than 30 seconds to build both the corpus (corpora for the two systems) with stereotypes and the corpus without stereotypes.

Names such as identifiers, function name, etc. are split according to the standard separators [25]. An underscore, '_', is used as a separator to split identifiers that contain more than one word, e.g., *feature_location* after splitting becomes *feature*, *location*, and *feature_location*. Camel casing is also used as a separator, e.g., *FeatureLocation* is split into *Feature*, *Location*, and *FeatureLocation*, and *FEATURELocation* is split into *FEATURE*, *Location*, and FEATURELocation.

The final step of preprocessing is partitioning the code into documents. Each function is considered to be a separate document (i.e., level of granularity). Typically, a document in the corpus can be a file of source code or a program entity such as a class, function, interface, etc. When the preprocessing is completed the software system is represented by a set of documents, $S = \{d_1, d_2, ..., d_n\}$, where d_i is any contiguous set of lines of source code and/or text. Each document d_i contains the function name, identifiers that the function uses, internal comments, string literals, and the stereotype annotation (if it has been re-documented) for each the function. After these steps, the corpus is constructed.

C. Indexing

The next step is to index the corpus using LSI. After creating the LSI space (using SVD), each document d_i in system S will have a corresponding vector v_i. Reduction of dimensionality is done in this step and reflects the most important latent aspects of the corpus. The dimension of the vector is a parameter of the algorithm. It is normally between 100 and 300 [6]. The typical manner to choose this value is to run experiments with different values (e.g., 100, 200, 300) and select the one that gives the best results with respect to evaluation measures as shown later [6].

Measuring the similarities between any two documents $sim(d_i,d_j)$, can be done by measuring the similarities between their correspondents vectors. Here cosine similarities are used. By studying and analyzing these similarities, we can identify the semantic information regarding source-code fragments, and the relations connecting them.

D. Formulating and Ranking Queries

The user formulates a query by using natural language to describe a change request in the same manner as [8]. A user query (q) is converted into a document of LSI space (d_c) and vector (v_q) for it is constructed. Based on the similarity measure between v_q and all documents in the corpus, the most relevant documents to v_q are retrieved as a ranked list $\{P_1, P_2, ..., P_n\}$.

Once LSI retrieves the relevant documents ranked by their similarities to the user query, then the user has the task of inspecting and investigating these documents to decide which of them are actually relevant to the query. The first ranked document (P_1) will be investigated first and then (P_2) and so on.

The user decides when to stop investigating. If the user discovers a part of the feature, then the intended feature is located successfully. Otherwise, the user can reformulate the query taking into account these results.

V. Experimental Study

A feature-identification study, over two open-source software systems is conducted to evaluate and compare the results of LSI and LSI+S. The study is designed based on recommendations from [30]. Both techniques, LSI and LSI+S, are applied independently and then the results compared. The only difference between the techniques is the inclusion of the stereotype information in LSI+S. Otherwise, the parameters used and the construction of the corpus is exactly the same. We chose one large and one medium-size open-source system to demonstrate the scalability/practicality of the proposed approach.

A. Design and Objectives of the Experimental Study

The first system is HippoDraw, an open-source application written in C++ that provides a data-analysis environment. It includes data-analysis processing and visualization with an application GUI interface, and can be used as a stand-alone application or as a python extension module. HippoDraw source code is well written and follows a pretty consistent object-oriented style. Its library consists of approximately 50K LOC and over 300 classes. HippoDraw 1.21.3 release is used in our study since it's well documented.

The second system used is the open source cross-platform application and UI framework Qt. It has extensive international support, as developers from Nokia, Digia, and other companies are involved in Qt's development. Qt is mainly written in C++ but has some language extensions with a special code generator (called the Meta Object Compiler) and special macros. It is cross platform for Windows, Linux, or Mac, and all of its editions support a wide range of compilers (e.g., gnu gcc, and MS Visual Studio). The Qt 4.4.3 release is used in our study. The major purpose of this particular release is to supply bug fixes and performance developments based on both internal testing and client feedback.

TABLE 2. DETAILS OF THE CORPUS USED AS INPUT TO LSI FOR EACH OF THE TWO SYSTEMS USED IN THE EXPERIMENTAL STUDY.

	HippoDraw 1.21.3	**Qt 4.4.3**
Vocabulary Size	6,803	91,187
Number of Parsed Documents/Methods	3,706	70,871
Dimensionality Used	200	300

Table 2 describes the characteristics of HippoDraw and Qt in the context of their use for LSI. It is clear that Qt is a much larger system in all aspects. We apply both LSI and LSI+S separately to each system. This allows us to compare the results and assess their quality relative to each other for the context of the added stereotype information. The method level of granularity is chosen in both studies. We followed the same methodology described in section IV in ranking the relevant parts of source code with respect to user query, with different dimensionality-reduction factors chosen for each study.

B. Evaluation Measures

To evaluate the results of feature location, a number of studies [7, 9, 12], use the position of first relevant method as an effort measure. Other studies [31] use recall and precision measures. In addition to these we compute the total effort measurement and use the position of the last relevant method. All of these measures as well as p-value are used to evaluate the results of the LSI and LSI+S approaches.

First we use the standard IR measurements [4] *recall* and *precision*. Recall of 100% means that all the relevant documents are recovered, though there could be recovered documents that are not relevant. Precision of 100% means that all the recovered documents are relevant, though there could be relevant documents that were not recovered. Typically there is a tradeoff between precision and recall. If recall is high, then precision normally is low. If precision is high, then recall normally is low. In computing recall and precision we only include the first 100 (cut point) ranked items produced by the query as was shown and justified in [32]. This is a standard approach to computing these values as anything more than 100 is beyond what a developer would normally investigate. Recall and precision are defined as follows:

- Recall = |relevant ∩ retrieved| ÷ |relevant|
- Precision = |relevant ∩ retrieved| ÷ |retrieved|

The main goal of all feature-location techniques is to reduce the effort of the developers in the location process. Therefore, in this evaluation, as has been done by previous researchers [4], we measure the effort that the developers need (maintenance-effort measurements) as the number of methods from the retrieved ranked list that they have to investigate until finding the first *relevant* method (PFR), the last *relevant* method (PLR), and all *relevant* methods (∑ EM). Typically, with respect to maintenance-effort measurements, lower values are preferred. These measures are defined as follows:

- ∑ EM: Total Effort Measurement (number of methods the developer needs to investigate to find *all* relevant documents).
- PFR: Position of first relevant document.
- PLR: Position of last relevant document.

We use the Wilcoxon signed-rank test to examine whether the difference in terms of effectiveness (**Σ EM** - Total Effort Measurement) for two approaches is statistically significant by computing the p-value.

C. Feature Selection & Determining Relevant Methods

For both systems in the study we selected 11 features for each (see Table 3 and Table 5). The features were selected based on bug reports in the online system documentation for both HippoDraw and Qt. A concatenation of the title and the description of the bug were used. Additionally, we conducted a

separate experiment for 14 more features found by inspecting eight bug reports in Qt.

Both systems have extensive and very complete documentation. Developers maintain very detailed bug reports and descriptions of the modification to fix each. We determined manually the set of relevant methods for each feature using this documentation as described below. For each feature we examined the related bug reports and descriptions of the fixes. We included all the methods that were modified in response to the bug fix and conducted a manual inspection of the code to determine all other methods relevant to that feature. Two graduate students did this inspection. They individually used the systems websites, bug-tracking reports[1], source code, etc. This collected data was then examined and any differences were resolved by additional inspection. This process took approximately 20 person/hours for HippoDraw and approximately 60 person/hours for Qt.

TABLE 3. HIPPODRAW FEATURE DESCRIPTION, APPLIED QUERY, AND THE NUMBER OF RELEVANT METHODS FOR EACH FEATURE.

Feature	Query	Number Relevant Methods
1. change font size	*change font size weight set*	10
2. change font style	*change font style italic*	18
3. update zoom mode	*update zoom mode zoomin zoomout*	9
4. reset printer settings	*reset change printer settings*	8
5. add item	*insert add item canvas*	7
6. remove item	*Delete remove item canvas*	7
7. change mouse property	*Option change mouse property*	9
8. change cut color	*change cut color set*	7
9. change representation color	*change representation color set*	7
10. make new display	*make new display add make*	12
11. update axis modeling	*update axis modeling reset*	8

D. Locating Features in HippoDraw

For version 1.21.3 of HippoDraw we ran our experiment on the 11 features and queries described in Table 3. For the corpus that was re-documented, we examined the stereotypes of relevant methods. It was found that all of the relevant methods for all features were labeled with at least one stereotype. That is, no relevant method was unclassified, which is a possible result from the re-documentation process. For overall distributions and details of the specific stereotyping of the HippoDraw system we refer the reader to [14].

In order to find the best user query that describes the intended feature accurately and completely, other researchers have used the process of formulating four different user queries and then choosing the best one among them [8]. The same procedure is followed here. For each feature in Table 3, the given query gives the best results of the four queries that were investigated. That is, the taken query gave the best ranking of the relevant documents for LSI. Table 3 also presents the number of relevant documents (methods) for each feature. With respect to dimensionality reduction for LSI and LSI+S, we determined 200 as the best value using the previously described method.

Table 4 summarizes the results obtained in identifying the features in the HippoDraw study. The first column indicates the feature number (from Table 3), the 2nd indicates the total effort measure, and the 3rd and the 4th columns indicate the positions of first and last relevant documents in the corpus respectively. As we can see in Table 4, using stereotypes (LSI+S) improved all three measures comparing with the result of using no stereotypes (LSI). The first relevant method (PFR) for LSI+S is equal or better to LSI. The precision and recall results are shown in Figure 2 and Figure 3, respectively. These figures show that LSI+S improves both recall and precision compared to LSI alone for most features. Specifically, the recall and precision improved for 9 features using LSI+S, while for 2 features the recall and precision are the same for both approaches.

TABLE 4. RESULT OF HIPPODRAW FOR THREE MEASUREMENTS; TOTAL EFFORT MEASUREMENT (Σ EM), POSITION OF FIRST RELEVANT DOCUMENT (PFR), AND POSITION OF LAST RELEVANT DOCUMENT (PLR).

Feature	Total Effort Measurement (Σ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	208	103	8	1	109	32
2	466	362	9	3	70	54
3	172	98	6	1	36	22
4	328	231	3	2	210	100
5	455	339	1	1	216	183
6	648	484	12	10	238	138
7	834	544	4	1	121	67
8	1595	764	2	2	1290	534
9	602	471	1	1	250	174
10	503	387	2	1	125	94
11	1721	843	3	1	1200	388

E. Locating Features in Qt

For version 4.4.3 of Qt we ran our experiment on the 11 features and queries described in Table 5. The same steps taken on the first system were also done here. Again, we chose four different queries, and then chose the best one among them. Experiments with different dimensionality reduction values showed that 300 gave the best results. Table 5 presents the summarization for all investigated features and the best queries used to locate these features. Table 6 summarizes the results obtained in identifying the features in the Qt study. As we can see LSI+S results in better values for all three measures compared with LSI alone. For this study, the precision and

[1] See https://bugreports.qt-project.org

recall results are also shown in Figure 4 and Figure 5 respectively. Again, LSI+S improves recall and precision.

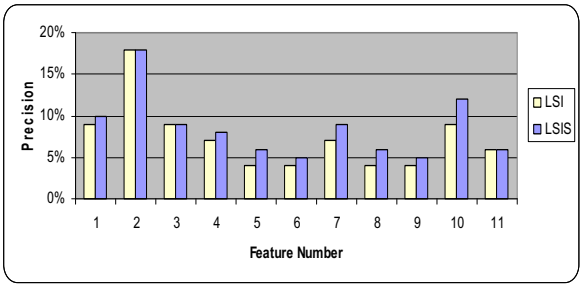

Figure 2. Precision results for the HippoDraw.

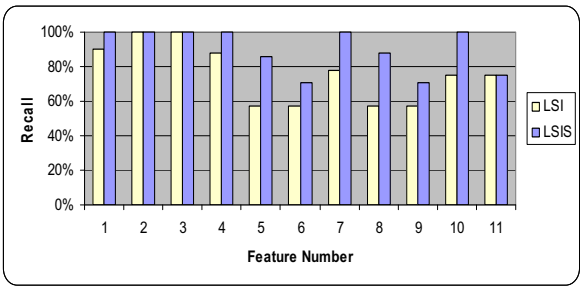

Figure 3. Recall results for the HippoDraw .

TABLE 5. QT FEATURE DESCRIPTIONS; FEATURE NAME, APPLIED QUERY, AND NUMBER OF RELEVANT METHODS TO EACH FEATURE.

Feature	Query	Number Relevant Methods
1. update font settings	*font update options settings reset*	21
2. create new font	*create new font*	24
3. change font size	*size font change*	23
4. set password	*set password change*	12
5. set RGB	*update RGB color RGBA RGBF*	7
6. add menu	*add create new menu insert menubar*	15
7. remove menu	*menu remove delete*	7
8. add action	*insert action add new*	11
9. remove action	*action delete remove*	9
10. search	*index search searching searcher indexing find*	12
11. draw polygon	*points polygon draw lines polyline*	7

In addition to the previous ones, we then examined an additional 14 features that were derived by investigating, eight new bug reports in Qt. These bug reports are given in Table 7. These 14 features were chosen because they were the most frequently changed. LSI+S improved or preserved the position of the most relevant method in each case. For instance, the bug 24685 affected versions 4.7.4 and 4.8.0, and was fixed in version 4.8.3. Based on the bug description, it occurs when the method *QPainter::drawText()* is called from a thread. A memory leak occurs if the text contains Russian characters (i.e.,"Время"). For this bug to be fixed the three functions *painter()*, *setFont()*, and *drawText()* all need to be modified. For the query we used the bug title "*memory leak in drawText()*". Using LSI these three methods were ranked 47, 65, and 11 respectively, while using LSI+S they were ranked 28, 31, and 1. An explanation for this result is that the function *drawText()* is overloaded 18 times, 9 of them have only one line of code in the body of the function, and were labeled with predicate or void-accessor. The others have different and more complex behavior, and were labeled as command-collaborator or void-accessor. In the context of our query the most relevant *drawText()* function is labeled with command-collaborator like the other two relevant methods *painter()* and *setFont()*. This function is ranked in the first position using LSI+S, while it is ranked in the 11th position using LSI alone.

Another example is the bug 11204, which impacts version 4.6.2, and is fixed version 4.7.1. Based on the description of this bug, this bug involves two features "*direction of text"* and "*alignment of text"*. Table 8 gives the relevant methods for this bug, and how they were ranked using both techniques. In this experiment we used the bug title "*direction change no longer implies alignment change*" as a query. The total effort measure for those new 14 features is examined, LSI+S has better values for all features with 38% average improvement. Moreover, the position of the most relevant method is improved using LSI+S for 10 out of 14 features, where for the remaining 4 features, LSI+S gives the same ranks as shown in Table 7.

TABLE 6. RESULT OF QT FOR THREE MEASUREMENTS; TOTAL EFFORT MEASUREMENT (EM), POSITION OF FIRST RELEVANT DOCUMENT (PFR), AND POSITION OF LAST RELEVANT DOCUMENT (PLR).

Feature	Total Effort Measurement (Σ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	2208	1846	2	1	1054	332
2	1900	928	1	1	520	467
3	1668	1192	4	1	684	443
4	1760	996	4	1	710	359
5	112	100	19	8	59	40
6	2792	1667	2	1	830	451
7	251	149	1	1	101	94
8	1239	701	3	1	815	456
9	359	185	1	1	153	100
10	1078	599	2	1	184	150
11	1641	566	1	1	1321	450

VI. DISCUSSION

Our hypothesis was that adding stereotype information to the corpus (source code) would improve the results of LSI in the context of the feature-location problem. It is quite clear

from the data that the addition of the stereotype information does improve the results of feature location using LSI for the presented queries in the context of these two systems. In all cases, and for all measures, LSI+S has equal or better values.

Examining the results of the studies, given in Table 4 and Table 6, we see that the position of the first relevant method improved with LSI+S in approximately 75% of the queries. The remaining 25% produced the same value. Moreover, the position of the first relevant method for LSI+S is in the first position in 7 of the 11 features for HippoDraw and 10 of the 11 features in Qt. Using LSI alone produced first positions of 2 of 11 for HippoDraw and 4 of 11 for Qt. This is a particularly nice improvement in the context of usability for the developer, as they need not look far into the list for a result.

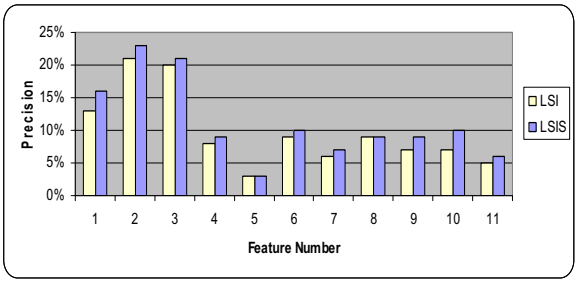

Figure 4. Precision for the 11 features from Qt.

Furthermore, the position of the last relevant method has been improved for all studied features in all cases with LSI+S. The improvement in this measure is much more evident (approximately one half on average). In

Table 9 we summarize the difference between the first and last relevant method positions for the two approaches for HippoDraw and Qt respectively. We see that there is an average improvement for these 11 features of 43% for HippoDraw and 36% for Qt in the distance from the first relevant method to the last relevant method.

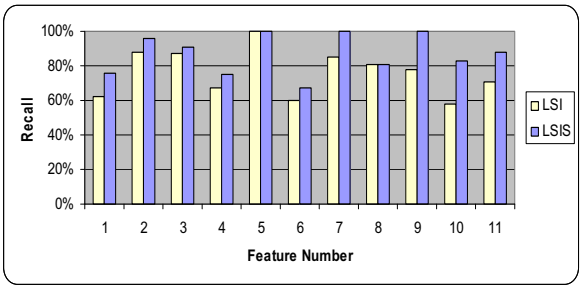

Figure 5. Recall for the 11 features from Qt .

The total effort measure is examined in Table 4 and Table 6. LSI+S again has better values for all queries. The average improvement is 46% with a range of 11% to 66% for both HippoDraw and Qt. From a usability standpoint this means that a developer would need to wade through far fewer methods on average to find all relevant methods.

TABLE 7. DESCRIPTION OF EIGHT BUGS (14 FEATURES) FROM QT BUG REPORTS..

Bug Number	Component	Number Relevant Methods	Rank of Most Relevant	
			LSI	LSI+S
24685 (1)	GUI: Font handling	3	11	1
15754 (3)	GUI: Font handling	7	6	3
11204 (2)	GUI: Text handling	4	3	1
5002 (2)	GUI: OpenGL	10	5	5
4210 (2)	GUI: Painting	9	7	4
2276 (1)	Widgets: Itemviews	13	11	9
1868 (2)	GUI: Text handling	8	1	1
935 (1)	GUI: Workspace	7	25	14

TABLE 8. RESULTS FOR LOCATING THE RELEVANT METHODS FOR BUG 11204.

Rank LSI+S	Relevant Methods	Rank LSI
1	direction()	43
262	setTextDirection()	285
5	setAlignment()	5
17	fixedAlignment()	21

TABLE 9. THE DIFFERENCE OF FIRST RELEVANT AND LAST RELEVANT METHOD FOR EACH QUERY IN HIPPODRAW AND QT. THE PERCENTAGE COLUMN IS THE IMPROVEMENT USING LSI+S.

Feature	HippoDraw			Qt		
	LSI	LSI+S	%	LSI	LSI+S	%
1	101	31	69%	1052	331	69%
2	61	51	16%	519	466	10%
3	30	21	30%	680	442	35%
4	207	98	53%	706	358	49%
5	215	182	15%	40	32	20%
6	226	128	43%	828	450	46%
7	117	66	44%	100	93	7%
8	1288	532	59%	812	455	44%
9	249	173	31%	152	99	35%
10	123	93	34%	182	149	18%
11	1197	387	78%	1320	449	66%
Average Improvement			**43%**			**36%**

The results for recall and precision for both studies are shown in Figure 2, Figure 3, Figure 4, and Figure 5. For both systems LSI+S has equal or better precision and recall values. Other studies that have used LSI alone [6] or combined with other analysis [2, 5] approaches produce comparable precision and recall values. This improvement appears to be on the same order as what has previously been observed.

The Wilcoxon signed-rank test was performed to investigate whether the difference in terms of effectiveness for the two approaches is statistically significant. We computed it for the (**Σ EM**) dependent variable. The null hypothesis is that there is no statistical significant difference in terms of effectiveness between LSI and LSI+S. The alternative hypothesis is that LSI+S has statistically significant higher

effectiveness than LSI. Our results for the two systems were found to be statistically significant. The p-value is lower than $\alpha = 0.05$ for the two systems, and was actually less than 0.0001. This allows us to reject the null hypothesis.

All the data from the three experimental studies supports the hypothesis that the addition of relevant information (in this case the stereotype) improves the results of querying in the context of feature location. This lays the foundation to generalize the result further, however we need to explain why this particular type of information helps. Beyond the abstract information-theoretic explanation (i.e., more information will give you better results) it would be prudent to understand some of the specific reasons that we see improvements.

It has been found that when using LSI, methods with small bodies and small numbers of identifiers are not ranked correctly [9] because there is not enough terms to properly build an accurate vector representation. However, the addition of stereotypes seems to mitigate this problem to some degree. That is, small methods appear to be ranked more correctly with the extra stereotype information. For example, in HippoDraw feature 3 "update zoom" using LSI resulted in the first relevant function *getZoomMode()* being ranked in the 6th position, while using LSI+S it is ranked first. We investigated this further and made some interesting observations. LSI ranked the function *hasZoomY()* in the first position, which is not relevant to the feature. However, *hasZoomY()* is small with only a couple lines of code. When re-documented, it is labeled with the *predicate* stereotype. This additional information changed the similarity between it and the query. We observed this same type of situation happening elsewhere. That is, small methods being ranked high by LSI but after being labeled with stereotypes receiving a much lower ranking.

We believe that using the stereotype information acts as a type of filtering mechanism when building the LSI subspace. That is, simple methods such as *get*/*set*, are superficially related to a feature, as they rarely impact the actual behavior and often play little part in the actual maintenance task. However, this belief is speculative in part and further investigation is needed to substantiate or generalize this hypothesis.

Stereotypes, by nature, increase the similarities between any two methods that have the same category. Since stereotypes are an abstract summary of a method's role and behavior, therefore, this implies that methods with similar roles will be made more similar (within the LSI subspace). Table 10 presents an overview of how the relevant methods were stereotyped. This is for both systems across all the 36 features. There were 311 relevant methods. We see that the vast majority (almost 90%) are labeled with the *command* and/or *collaborator* stereotypes. Approximately 6% are predicates and the remaining is a variety with no single stereotype category making up more than 5.4%. In short, the most relevant methods, in these three studies, are almost always some type of *command or collaborator* method. We observed this distribution after running the studies while attempting to better understand the results.

Command and *collaborator* methods do the majority of the logic within a class. They model the behavior of a class and hence provide most of behavior of observable system features. Thus, it makes sense that the most relevant methods for any system feature would most likely be of the *command* stereotype.

TABLE 10. DISTRIBUTION OF STEREOTYPES FOR THE RELEVANT METHODS OVER BOTH STUDIES. THE OTHER 17 WERE A VARIETY OF DIFFERENT STEREOTYPES WITH NO ONE CATEGORY MAKING UP MORE THAN 2%.

Stereotype	Number of Methods	Ratio (%)
Command-Collaborator	221	71%
Command	53	17%
Predicate	20	6%
Others	17	6%
Total	**311**	**100%**

VII. THREATS TO VALIDITY

A number of issues could affect the results of the study we conducted and so may limit the generalizability of the results. The authors attempted to minimize factors so to decrease their effect. Feature selection is an issue. We picked features that were commonly modified in the systems based on the documentation. We also needed features for which all relevant methods could be identified. As such they were selected with no preconceived notion of how well either LSI or LSI+S would perform on them.

The number of queries we used could also be too few for a rigorous comparison. Compared to other studies on feature location [8] the number we used, 30 queries over 36 features, represents a bit larger set (i.e., previous studies have used as few as three queries). However, other studies [33] have used more but they depend on bug reports titles or descriptions directly as a query without filtering or preprocessing. They also only include items that were changed due to the bug report. This may not include all relevant items, but only relevant items that were changed. Another issue is if the features used in this study are representative to those used in practice. Taking features directly from active open-source systems minimizes this to a degree. Also, these features were involved in actual maintenance tasks. We also minimized this threat by selecting two different systems from two different domains. Expanding the study to other systems could further minimize this issue. Another issue is that query selection depends on the knowledge of the user. We attempted to minimize this by selecting the best query for LSI from the set of four queries.

Lastly, we may not have found all relevant methods or may have labeled methods as relevant that actually were not. This was addressed by a careful manual inspection of the systems and associated documentation.

VIII. CONCLUSIONS AND FUTURE WORK

A novel technique to improve the results of using Latent Semantic Indexing on the problem of feature location is introduced. The technique involves adding new information to the source code before applying LSI. In this case, the new information added is method stereotypes, which were derived via static program analysis from the source code.

We compared the results of using LSI on the original code base with that of a version re-documented with stereotype

information. This experimental study on two open-source systems demonstrated that the added stereotype information improved the query results for the feature-location process. We saw substantial average improvements in the results for all measures. For each individual query we saw equal or better results in all cases when using the stereotype information. The results were compared using recall, precision, position of first and last relevant document, and a total effort measure.

The implications of these results are important for a number of reasons. The results confirm that adding information to a corpus (here source code) will improve the results for extracting and querying that corpus. The results provide evidence that the addition of other information than stereotypes, gained via static or dynamic analysis of the code, could also improve the results. The results also imply that stereotype information is relevant for feature location (and comprehension), which supports our previous studies on stereotypes. This last issue could give rise to a new means for evaluating techniques to support comprehension. If we claim that adding or deriving particular information from source code supports comprehension, then it should in theory also improve the results of IR methods such as LSI.

Future work on this topic includes a better understanding of why stereotypes improve the results. Additionally, we are investigating what other types of information added to source code improves the results of IR methods.

REFERENCES

[1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," CACM, vol. 37, pp. 72-82, 1994.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," JSME, vol. 25, pp. 53-95, 2013.

[3] R. J. Turver and M. Munro, "An early impact analysis technique for software maintenance," *JSME*, vol. 6, pp. 35-52, 1994.

[4] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution," in *Ency. of SE*, 2010.

[5] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," TSE, vol. 29, pp. 210-224, 2003.

[6] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in WCRE, 2004, pp. 214-223.

[7] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *TSE*, vol. 33, pp. 420-432, 2007.

[8] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in ASE, 2007, pp. 234-243.

[9] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," in ICPC, 2006, pp. 137-148.

[10] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in ICPC, 2007, pp. 37-48.

[11] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in ICSE, 2001, pp. 103-112.

[12] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in ICPC, 2010, pp. 14-23.

[13] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev, "IRiSS - A Source Code Exploration Tool," in ICSM, 2005, pp. 69-72.

[14] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in ICSM, 2006, pp. 24-34.

[15] A. Perotte, N. Bartlett, N. Elhadad, and F. Wood, "Hierarchically Supervised Latent Dirichlet Allocation," in *Neural Information Processing Systems*, ed, 2011.

[16] H.-M. Müller, E. E. Kenny, and P. W. Sternberg, "Textpresso: An Ontology-Based Information Retrieval and Extraction System for Biological Literature," in *PLoS Biol.*, 2, e309, ed, 2004.

[17] J. Teevan, "Improving Information Retrieval with Textual Analysis: Bayesian Models and Beyond," Master's thesis, Massachusetts Institute of Technology, 2001.

[18] D. C. Blair, "Information retrieval and the philosophy of language," Review of Info Science and Tech, vol. 37, pp. 3-50, 2003.

[19] T. W. C. Huibers, M. Lalmas, and C. J. v. Rijsbergen, "Information retrieval and situation theory," SIGIR Forum, vol. 30, pp. 11-25, 1996.

[20] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in ICSM, 2010, pp. 1-10.

[21] D. Poshyvanyk, "Using information retrieval to support software maintenance tasks," in ICSM, 2009, pp. 453-456.

[22] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," in IWPC, 2000, pp. 241-249.

[23] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *JSME*, vol. 7, pp. 49-62, 1995.

[24] E. Wong, S. Gokhale, and J. Horgan, "Quantifying the closeness between program components and features," *JSS*, vol. 54, pp. 87-98, 2000.

[25] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in ICPC, 2009, pp. 218-222.

[26] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Non-Interactive Approach to Feature Location," in ICSE, 2004, pp. 293-303.

[27] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in SCAM, 2011, pp. 173-184.

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

[29] G. Salton and M. J. McGill, Introduction to Modern Information Retrieval. : McGraw-Hill, 1983.

[30] R. K. Yin, *Case Study Research: Design and Methods (4th Edition)*. Thousand Oaks, CA: Sage, 2009.

[31] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in ICSE, 2011, pp. 111-120.

[32] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimothy, "Using information retrieval based coupling measures for impact analysis," *EMSE*, vol. 14, pp. 5-32, 2009.

[33] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *TOSEM*, vol. 21, pp. 1-34, 2013.