Static Techniques for Concept Location in Object-Oriented Code

Andrian Marcus, Václav Rajlich, Joseph Buchta, Maksym Petrenko, Andrey Sergeyev

Department of Computer Science
Wayne State University
Detroit, MI 48202
{amarcus, rajlich, joe, max, andrey}@wayne.edu

Abstract

Concept location in source code is the process that identifies where a software system implements a specific concept. While it is well accepted that concept location is essential for the maintenance of complex procedural code like code written in C, it is much less obvious whether it is also needed for the maintenance of the Object-Oriented code. After all, the Object-Oriented code is structured into classes and well-designed classes already implement concepts, so the issue seems to be reduced to the selection of the appropriate class. The objective of our work is to see if the techniques for concept location are still needed (they are) and whether Object-Oriented structuring facilitates concept location (it does not).

This paper focuses on static concept location techniques that share common prerequisites and are search the source code using regular expression matching, or static program dependencies, or information retrieval. The paper analyses these techniques to see how they compare to each other in terms of their respective strengths and weaknesses.

1. Introduction

Searching in source code or documentation is one of the most common activities performed by software engineers during maintenance [27]. Concept location is one such searching activity where the software engineers try to locate a part of the source code that implements specific domain concepts. This activity is also referred to as the concept assignment problem [3].

Concept location occurs frequently during incremental change of software [24]. Here, concepts are extracted from change requests while the concept location process identifies the starting point of the change.

Needless to say, the complexity and importance of the concept location process increases with the size of the software. The goal of concept location techniques and tools is to reduce the search space that the developer needs to investigate. Different techniques achieve this in different ways. One common feature of various approaches is that the source code is often decomposed into units different than files (e.g., classes, functions, etc.) and it is enriched with additional information (e.g., relationships between elements of the source code). The software decomposition determines the unit of the search, while the additional information determines the searching criteria.

Software decomposition and analysis creates an intermediate representation of the software system, see Figure 1. Deterministic mappings are defined between this representation and the source code. The user interacts with and searches this representation, but the results of the search are presented as elements of the source code.

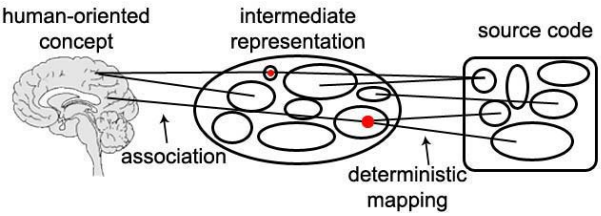

Figure 1. Most concept location techniques rely on an intermediate representation of the source code.

At a high level, the concept location process can be defined as follows: (1) concept formulation (usually in natural language); (2) query formulation and execution based on the intermediary representation; (3) investigation of results. Most concept location techniques have the following attributes:

1. Prerequisites (e.g., complete and executable program, test suite, incomplete program, libraries, etc.);
2. Intermediate representation:
   a. Format (e.g., string, graph, database, etc.);
   b. Content (e.g., text, dependencies, data flow, control flow, execution traces, etc.);
   c. Preprocessing/analysis (e.g., manual, automatic, dynamic analysis, static analysis, parsing, knowledge base, etc.);

Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)
1092-8138/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

3. Query formulation
4. Format and granularity of results (e.g., line of text, line of code, function, class, file, etc.).

Based on the preprocessing needed to create the intermediate representation we can differentiate two major classes of techniques: static and dynamic. Static techniques create the intermediary representation based on the source code that can be incomplete. Dynamic techniques require complete executable programs and test suites.

All static techniques share the same prerequisites and have compatible preprocessing; therefore this paper is focused only on three of the most popular static techniques based on regular expression matching, static program dependencies, and information retrieval.

While it is well accepted that concept location techniques are necessary for the maintenance of complex procedural code like one written in C, it is much less obvious whether they are needed for the maintenance of Object-Oriented (OO) code. After all, the OO code is structured into classes and well-designed classes implement singular concepts, so the issue seems to be just to select the appropriate class. The objective of our work was to see whether the techniques of concept location are still needed in OO code, and whether OO structuring facilitates concept location. We also compare the techniques to each other and gather information about their validity.

The paper is organized as follows: Section 2 covers related work; Section 3 presents the three static techniques for concept location; Section 4 presents a case study; and Section 5 presents our conclusions and directions for future research in this area.

2. Related Work

One of the oldest and most widely used searching techniques available to programmers is the pattern-matching approach. This technique is supported by a family of tools such as grep, egrep, fgrep, ed, sed, awk, and lex, and it performs the search through pattern matching on character strings.

Since these operations have a low granularity, the structural information embedded in the source code is ignored [23]. A series of tools such as SCRUPLE [22], LSME [21], and GENOA [7] address the problem of retaining structural information by introducing a pattern specification language to specify high-level patterns that reflect programming language constructs. LSME [21] allows the user to express what information to extract from the source code as lexical specification (regular expression with lexical tokens). Based on such a specification, the system produces a source model without parsing the code. The GENOA framework [7] is a programming language specific tool, which analyzes the source code and provides a scripting language to specify the analysis.

Another approach to searching (or browsing) the source code is to represent the system as a graph in which the nodes represent some entity of the system and edges represent relationship between the entities. Rigi [20], Scan [23], the environment described in [10], FEAT [25], and Ripples [4] are tools based on this approach. Ripples uses dependency graphs to aid the user during the process of incremental change [24], whereas FEAT facilitates feature separation using concern graphs. The strength of graph-based techniques is the ability of graphs to retain structural information (e.g., inclusion, control-flow, data-flow, inheritance, etc.) that is very important during searching and analysis of relevant results. The software engineers search this graph for the location of the concept. They need to have information on relationships among the items they are analyzing [27].

Semantic information that is embedded in the comments and identifiers in the source code is specifically targeted in concept location techniques based on information retrieval (IR). IR systems [26] are used to store, manage, and retrieve (usually unstructured) data such as text. They are widely used in libraries, web search engines, etc.

IR methods are also used in concept location. Differences among approaches are in the format of the intermediate representation (i.e., the signature), the preprocessing needs, and granularity of results. Some tools use vector space or probabilistic models for the intermediate representation [1, 17], others use more sophisticated representations [18]. Some tools use the identifiers alone in the intermediate representation [1], while others also include the comments and external documentation [18]. Finally, some tools work on function or class level [1, 18], while others work at component level [17]. Some approaches employ natural language processing of the source code [1] and some require parsing of the source code [1, 18].

Other concept location techniques are based on creating a knowledge base (or domain model) that represents the concepts of the software domain. Once the domain model is created, it is used to locate concepts in the software modules. These techniques require considerable user involvement during the creation and maintenance of the domain model. IRENE [13], DM-TAO [2, 3], and HB-CA [11] are three different concept assignment systems that employ a knowledge base approach.

The technique described by Liu and Lethbridge [16] extract the search information from both the programmers' queries and the source code. Michail [19] emphasizes the importance of the information stored in the GUI of an application as the base for search and exploration.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

Table 1. Features of the static concept location techniques

	Pattern matching	Dependency based	IR based
Prerequisites	none	none	none
Internal representation format	string of tokens	graph	document vector space
Internal representation content	characters/tokens	class dependencies	identifiers, comments
Internal representation analysis	none	static dependency analysis	parsing; LSI
Query	regular expressions	depth first search in graph	natural language
Results	lines of text	classes	classes/methods

The intermediate representation used by the dynamic techniques is gathered by obtaining and analyzing execution data. Wilde developed the Software Reconnaissance method [30], which utilizes execution traces to locate features in existing systems. Wong et al. [31] analyze execution slices of test cases to the same end. Eisenbarth et al. [9] use both dynamic information gathered from scenarios of invoking features in a system and also static dependencies. They extend the Software Reconnaissance approach to detect multiple features, using concept lattices. Harman et al. [12] propose syntax-preserving static slicing in order to refine and extend the results of concept assignment.

Little work has been done on comparing various concept location techniques. Wilde et al. [29] compared static and dynamic techniques on a procedural software. Also very little work has been done on concept location in OO systems.

3. Static Concept Location Techniques

One common characteristic of static location techniques is that they can be used rapidly without too much preparation. They allow work on incomplete programs, design documents, and other work-products. This allows programmers to combine them and leverage their respective advantages. Table 1 summarizes the specific attributes of each technique. For methodological reasons we deal with each technique separately.

3.1. String Pattern Matching Technique (grep)

Grep is an acronym for "global regular expression print". It is a tool that prints out lines that contain a match for a regular expression. Even though there are several more advanced pattern matching tools, grep is one of the most popular tools within this category; therefore, grep can be taken as the representative of this searching tools class. We refer to the string pattern matching technique as grep-based technique.

Usually, work with grep-like tools involves the following steps:

1. Formulate the search pattern based on the concept, and execute the grep search.
2. If the size of the result is too large, then either:
   a. Refine the search pattern and execute the next search within the current results.
   b. Formulate a new search pattern and execute the search. This activity should be also performed if there are no results at all.

   Repeat step 2 until the size of the search results is reasonable.
3. If the current search result has a reasonable size, then review it to decide whether it contains the concept. Based on this decision the next step can be either:
   a. If the result contains the concept then determine its place and stop the search.
   b. If the current search result contains only the part of the concept then this result can be used to formulate a new, more precise search pattern.
   c. If the search result is completely irrelevant then formulate the new search pattern and repeat the search.

The critical part of the "grepping" process is the formulation of the search pattern. This can be facilitated by certain heuristics and in most cases it is highly dependent on the experience of the person who performs the search. The technique does not make any assumptions about the structure of the software and hence it can be used on OO systems without any adaptation.

In this paper, grep 2.5.1 from the CygWin tools package [5] was used. This version of grep is able to search the specified files and, as the search output, provide programmers with the list of lines that contain required pattern.

3.2. Dependency Search Technique

The static dependency search is a variant of the depth first search, conducted by a programmer rather than a computer. The programmer follows the dependencies among the modules, hence the technique is adapted to OO programming by dealing with classes and their dependencies. For example in Java, class A depends on B if class A refers to class B in a definition of a data member, local variable, argument, data cast; if A refers to class B's static members; if A inherits from B; or if A implements interface B. Similar dependencies can be defined for other programming languages. If A depends

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

on B, then A is *dependent* class (of B), while B is *supporting* class (of A).

When searching for concepts, the functionality of the classes that the programmer encounters can be viewed in two different ways. First, there is the *composite functionality* that is defined as the complete functionality of a class combined with all its supporting classes. The second type of functionality, *local functionality*, consists of concepts that are actually implemented in the class and are not delegated to others.

These notions can be illustrated via a class that implements a list of personal records. The class *List* has the composite functionality of both the list and personal records, while the local functionality is that of managing the linked list only; dealing with the records is delegated to a supporting class *Record*.

The role of the developer is to follow the dependencies and make decisions that direct the search. As the software is searched, the programmer has to do the following:

1. Select the starting point for the search. The top class (i.e. the one that contains main() or init()) in the call graph of the system) is a good candidate, because its composite functionality is the functionality of the entire system. If multiple classes contain function main(), the one whose composite functionality most likely contains the concept is selected as the start.
2. Review the chosen class to determine if its local functionality contains the concept. In doing so the source code and documentation are explored.
   a. If the local functionality contains the concept then the search is finished;
   b. If the concept is not present in the local functionality but it is present in the composite functionality, then choose one of the supporting classes and continue with step 2.
   c. If a wrong guess has been made and the concept is not present in the composite functionality, then backtrack to the previous class and begin again at step 2.

Step 2.b requires the programmer to find the supporting classes. Supporting classes are usually easy to identify because their names appear explicitly in definitions of dependent classes via local variables etc. However, a tool that extracts dependencies is useful in that it alleviates the programmer from this tedious and time consuming task.

If the search progresses without backtracks, then the number of steps is equivalent to the length of the path from the top class to the location of the concept. That usually involves only a small fraction of the classes of the system. In the worst case, it is equal to the depth of the call graph which again in most systems is much smaller than the total number of classes. This fact ensures that the entire system needs not be searched in order to locate a specific concept.

3.3. IR-based Technique (LSI)

IR-based methods for concept location share the following general pattern:

1. Preprocessing of the source code and documentation.
2. Indexing that creates the intermediary representation.
3. Execution of queries formulated in natural language.
4. Retrieving and analyzing the results that are returned as a ranked list.

In this paper we use the IR based system described in [18]. It uses latent semantic indexing (LSI) [6] for the intermediate representation for the identifiers and comments extracted from the source code. The source code is partitioned into a set of documents. A document can be any contiguous set of lines of the source code, therefore different document definitions are possible. The searching methodology requires a use of LSI tool [18] and the steps of the methodology are as follows:

1. Select a set of words that describe the concept. This set of words constitutes the *query*.
2. If a word of query does not occur in the software, then:
   a. Look up similar words using the dictionary of the software system (provided by the IR method). Decide whether to add these words to the initial set or not.
   b. Eliminate the word that is not part of software system's dictionary from the initial set. If the elimination of a word radically alters the perceived meaning of the query, go to step 2 and select additional or new words for the query.
3. Run the query using LSI. The tool will return a list of source code documents that are ordered based on the similarity to the initial query.
4. Examine source code documents from the list in the order they appear. Decide whether the currently examined document is part of the concept; if it is, then stop. If it is not and the new knowledge helps to formulate a better query, go to step 2. Else examine the next document from the list.

4. Case Study

The objective of exploratory case studies [32] is to study phenomena not sufficiently studied before. Our exploratory case study was to see whether the static techniques for concept location are still needed in OO code, whether OO structuring facilitates concept location, and how do these techniques compare to each other.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

4.1. Case Study Design

Our hypothesis is that the concept location techniques are still needed in spite of the OO structuring of the software, that OO structuring does not provide any particular advantage to concept location, and that each technique has unique strengths and weaknesses.

In order to broaden the insight provided by the case study, we conducted one part of it on Java software and the other part on C++ software system. We selected open source programs for the case study because of their easy accessibility, and the opportunity to replicate the case study if any other group decides to do so. In order to use typical change requests, we selected the change requests from the wish lists that are available from the software web sites. The wish list contains change requests for the future evolution of the software.

Three programmers worked on the case study in parallel, each of them employing a different location technique. Each of them extracted a concept from the change request in a way best suited for the particular location technique. In the end, we debriefed the programmers about the process and its strengths and weaknesses, and summarized the findings.

We also measured effort required by each technique. Each participating programmer was instructed to keep track of the number of lines of code that were read and comprehended within each technique. By choosing this measure, we assumed that most of the programmer time is spent in reading and comprehending code, while other activities like executing queries constitute a minor part. Other measures, such as time spent in the task, were considered but were abandoned as they may be more indicative of the programmers' experience rather than the complexity of the task.

In order to validate correctness of the concept location, the programmers always implemented and tested the required change as the last phase of their effort.

4.2. AOI Part of the Case Study

The first part of the case study was conducted on a software called Art of Illusion (AOI) [8]. AOI is a 3D modeling studio that supports rendering and animation. AOI is written in Java and it has 442 classes, 20 interfaces, and 100,838 lines of code.

The change request asks to implement a zooming control that uses both mouse or arrow keys for both zoom-in and zoom-out. Currently, the only way zooming is controlled in AOI is through a text box where a value of the zoom has to be typed in by the user; the default value is 100%.

A quick look at the classes of the program revealed that there is no class that would deal with zoom, whose simple update would solve the problem. Neither is there any obvious class that would subsume zoom functionality; therefore the concept location techniques must be used in order to locate the concept.

4.2.1. Location using grep. The search was performed step-by-step following the technique of Section 3.1.

Based on the change request, the *zoom* string was the first pattern to search for. The query produced 6 lines. Unfortunately, these lines were irrelevant to the oncept sought.

The second search was based on the same idea as the first search, but with a different word - *scale*. This pattern returned in 1,544 lines. This was deemed too large for inspection.

In order to make the search space smaller, we performed another search within the results of the previous search. For that, the AOI user interface was explored and it was found that the only GUI feature connected with the searched concept was the text box that allows the user to change the zoom scaling value. The default scaling value is 100. Therefore, a new query was formulated – *100*. This query returned 4 lines from the ViewerCanvas.java file.

While exploring the ViewerCanvas.java file we determined that the names of the methods assigned to the scale change and the current scale value are scaleChange and scaleField.Value respectively. Since the VieverCanvas.java class responds to some canvas actions, it was concluded that it is likely to also deal with related mouse and keyboard events.

During this concept location process approximately 60 lines of code were analyzed.

4.2.2. Location using dependency search. The static dependency search was employed using the approach described in Section 3.2.

First, the concept to be searched for was formulated (i.e., zooming within a scene). After executing the program it was observed that this concept is implemented via a text box, in which the user specifies the zoom value.

The search started at the ModelingApp class; this class contains the function *main*(). After analyzing this class it was determined that the concept was not contained within its local functionality. Figure 2 shows a class diagram of the classes that were searched, while Table 2 shows the size of these classes and the number of lines of code investigated in each class.

The next step was to search the classes that support the ModelingApp class. The LayoutWindow class was chosen to be searched since it was responsible for constructing the main AOI window. After searching the LayoutWindow class we determined that the composite functionality contains the concept, but the local functionality does not.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

There were several clues to also search SceneViewer class, which were given from the program comments, such as "displays the views of a scene". There were also clues to search ValueField class because this class implements the text box.

After analyzing the ValueField class we determined that the desired concept is not present in the composite functionality of ValueField. The search backtracked to the LayoutWindow class and SceneViewer class.

Table 2. Classes investigated in AOI using the dependency search method

Class Name	Total LOC	LOC Investigated
Modeling Window	925	50
ValueField	230	50
Layout Window	2431	65
SceneViewer	608	100
ViewerCanvas	2086	100

The SceneViewer class was explored and several functions were located that were responsible for responding to events from the user. For example, the function updateImage() was responsible for repainting the screen after one of the image properties had been changed. Based on this evidence we determined that the composite functionality of this function contained the concept.

Since the local functionality of SceneViewer still did not contain the concept, the next logical choice was to search its parent class, on which it depends, ViewerCanvas.

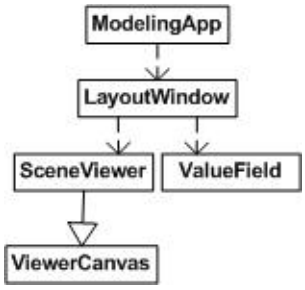

Figure 2. Dependencies among the investigated classes in AOI

After visiting ViewerCanvas several functions were located that respond to user commands and build the GUI interface. These included the function buildChoices() and scaleChanged(). The former is responsible for constructing the GUI and the later is responsible for updating the image after the user specifies a new zoom value. The concept was located within the ViewerCanvas class.

4.2.3. Location using LSI. From past experience, it was known that zooming sometimes is implemented through scaling in graphics packages. With this information, the following query was formulated: *zoom view scale*. The word *zoom* was not part of the software and following the methodology of Section 3.3, the query was modified to *view scale views*. The LSI tool produced a ranked list of classes. The first four classes (see first part of Table 3) did not contain the concept.

Table 3. Classes investigated in AOI using LSI

Rank	Class name	Total LOC	LOC investigated
	First query		
1	MoveViewTool	170	15
2	ScaleObjectTool	575	40
3	RotateViewTool	203	5
4	ScaleShiftModule	145	35
5	Camera	691	150
	Second query		
1	RenderingDialog	254	10
2	RenderSetupDialog	212	15
3	ObjectPreviewCanvas	113	5
4	ViewerCanvas	2086	170

The fifth class, Camera, indicated presence of closely related concepts and comprehension of this class allowed us to formulate second query that contained words *setscreenparams setscreenparamsparallel setdisttoscreen*.

The tool produced a new list of ordered classes, which were examined in the suggested order (see second part of Table 3). The RenderingDialog class implements the dialog box in which the user can watch a scene being rendered. The RenderSetupDialog class implements the dialog box in which the user can select a renderer and specify options on how a scene should be rendered. The ObjectPreviewCanvas class is used for previewing a particular object. Finally ViewerCanvas class was identified as containing the concept.

4.3. Doxygen Part of the Case Study

We conducted a second part of the case study on a software called Doxygen [28]. Doxygen is a Javadoc-like documentation system for C++, C, Java, and Objective-C. It can generate HTML, text, or Latex documentation from documented source files. Doxygen is written in C++, it has 117 classes and 232,199 lines of code.

The change request asks to implement a feature that will allow the class hierarchy to be displayed in alphabetical order according to namespaces. Currently, the ordering of the class hierarchy is alphabetical according to class names.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

A quick look at the classes of the program revealed that there is no class that would deal with class hierarchy, whose simple update would solve the problem. Neither is there any obvious class that would subsume this functionality; therefore the concept location techniques again must be used in order to locate the concept.

4.3.1. Location using grep. The concept that we selected was the concept "sort" and therefore we searched for *sort* string. This query returned 87 matching lines.

An additional search for the word *namespace* within the results of the previous query, returned 3 matching lines and showed that Doxygen does have a sorting function that uses namespace information (string *config.cpp: "sorted by fully-qualified names, including namespaces. If set to \n"*). The corresponding line in the file config.cpp was examined and it was learned that this function can be turned on by the option variable *SORT_BY_SCOPE_NAME*.

The next search involved the pattern *SORT_BY_SCOPE_NAME* and resulted in 6 lines. Analysis of these lines showed that they are involved in the class members sorting, but not in the class hierarchy sorting.

Since we needed to sort the class hierarchy, the next search string was *Class Hierarchy,* which produced 6 lines. Among them, the writeGraphicalClassHierarchy function is responsible for the graphical class hierarchy output, but not for the text output. This query led to the idea to search for *ClassHierarchy* as one word without the space.

Using the results of the previous query, a search for the *writeClassHierarchy* string (which is a function name) was done and 3 resulting lines were identified, all within the index.cpp file. The next analysis showed that the writeClassHierarchy function is responsible for the text output of the class hierarchy and can be easily modified to order class hierarchy based on the namespace information; therefore, this function is the place of the required concept.

Approximately 125 lines of code were investigated in this process. The queries produced 107 lines of matching source code lines. Not all of these lines were analyzed, but additional lines in the related source code were reviewed.

4.3.2. Location using dependency search. After reviewing the change request, it was concluded that the concept to be located was class hierarchy output.

The search began with the main.cpp file since it contained the main() function, see the class diagram in Figure 3. The main.cpp file of this program is very small and only contained a single dependency via function calls to the doxygen.cpp file. Therefore it was decided to continue the search with both the doxygen.cpp and the doxygen.h file. The header file was analyzed first because it contained a summary of the code contained in the implementation file (i.e., function headers, data types, etc.).

It was determined that the supporting class ClassSDict is likely to have the composite functionality that contains the concept. Since the local functionality of ClassSDict did not contain the concept, the parent class SDict was investigated. Once this class was visited, the programmer concluded that an invalid path had been taken and the search backtracked to doxygen.h.

The programmer decided to explore the function generateOutput() of doxygen.cpp in greater detail since it was called by the main() function. This function calls another function generateClassDocs(), which was also contained in the doxygen.cpp file. It was determined that the composite functionality of this function contains the concept. The statistics of the search are in Table 4.

Table 4. Classes and files explored in Doxygen using the dependency search method

Class/File Name	LOC	LOC Investigated
main.cpp	40	10
doxygen	8734	80
ClassSDict	160	50
Sdict	152	50
index	3371	170

The generateClassDocs() function calls the writeHierarchicalIndex() function that is contained within the index.cpp file. After a careful analysis of the function writeClassHierarchy(), it was determined that the function is responsible for displaying the class names from the source files. Therefore the programmer determined that the local functionality of this function contained the concept. With this information the search was concluded since the concept had been located successfully.

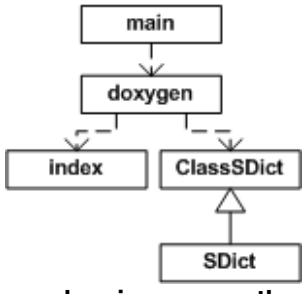

Figure 3. Dependencies among the investigated classes and files in Doxygen

4.3.3. Location using LSI. The initial query consisted of the words *class hierarchy list sort,* expanded by plurals and other grammatical variants.

Since Doxygen is written in C++ and has many global functions, the function granularity for the results was

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

used. After running the query, the returned functions were examined in the order suggested by the tool (see first part of Table 5). The first document examined was a global function initBaseClassHierarchy (from the util.cpp file), which initializes a data member for a list of classes. There was not enough information to determine that this function is a part of the concept and it was decided to examine the subsequent documents. During the analysis of the second function classHasVisibleChildren (from index.cpp), we gained a sufficient information to form a new query that consisted of words *writeclasstree hierarchy*. Each term in the new query was part of the Doxygen's dictionary.

After running second query (see second part of Table 5), it was concluded that the function writeClassHierarchy contains the concept that would have to be changed while the functions ranked ahead of it do not. The change request was successfully implemented in order to verify whether the function writeClassHierarchy is part of the desired concept.

Table 5. Functions explored in Doxygen using LSI

Rank	Function Name	LOC	LOC investigated
	First query		
1	initBaseClassHierarchy	12	10
2	classHasVisibleChildren	13	20
	Second query		
1	writeGroupHierarchy	10	5
2	writeHierarchicalIndex	48	15
3	writeDirHierarchy	10	5
4	writeGraphicalClassHierarchy	39	15
5	initClassHierarchy	9	9
6	countClassHierarchy	11	1
7	writeClassHierarchy	27	27

4.4. Discussion of the Case Study

During the case study we noticed that the concepts we searched for do not have an obvious location in a class hierarchy. Of the three location techniques used, the dependency search technique utilizes the class structure to the largest degree; even there the programmer occasionally took a wrong turn in the search and had to backtrack. Hence, we can conclude that the OO structure of the software is not always sufficient to allow programmers to easily and unerringly select the appropriate class as the concept location and thus, the use of the concept location techniques is necessary. Our explanation of this fact is that while concepts related to the change requests are potentially numerous, only a minority of them is implemented as specific classes and hence for all remaining concepts, concept location techniques still have to be employed.

This observation is also supported by research in the aspect oriented programming (AOP) [15]. It is accepted there that there are many concerns (or concepts) in software, which are not successfully captured by classes in OO systems. Mechanisms to encapsulate such crosscutting concerns are defined and used in AOP languages such as AspectJ [14]. The problem though still remains unsolved for legacy systems, written in languages that do not have aspect oriented features.

Another observation that we made is the fact that compared to other structuring principles, OO structuring does not provide any advantage for concept location. Among the techniques we investigated, grep technique provides output as lines of code and is completely independent of software structuring. LSI technique is based on partition of the code into documents and a document is any contiguous part of the code, hence it again is independent of software structuring.

Dependency search technique is based on ability of the programmer to recognize composite and local functionality and as long as this is guaranteed, the technique is applicable. In the past, the technique was used in well structured functional decomposition and worked well [4], while in a poorly structured code it worked poorly [29]. Hence it is the quality of the structuring, not a specific brand of structuring, that facilitates dependency search, and OO structuring did not offer any particular advantage over other structuring principles.

Another observation is that the concepts may be delocalized among several classes, for example they may be distributed between GUI and program logic. Both concepts that we searched for were delocalized in such a way and had a presence in both GUI classes and the program logic classes. To illustrate this situation by a simple example, consider a program that displays the user name. Modifications of such concept can be done in three possible ways:

1) By modification of the input data
2) By modification of the data, stored in the system
3) By formatting of the output data

Moreover, if the concept is implemented as a data container and has a long data flow path, the programmer can modify this concept at any step in the data flow path.

In this vein, our case studies discovered that there are at least two possible ways to implement the requested change in AOI: either by changing class ViewerCanvas, which is responsible for the GUI part of zooming, or by changing the scaleChange function which is responsible for the image updating after rescale operation. Nonetheless, even though both implementations are possible, changing the ViewerCanvas class is more natural and easier to implement, as it was discovered

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

during the consequent implementation phase. However, this multiple presence of the concept in the code was a distracting experience for all three techniques and yielded wrong searching paths.

We also noted that the static concept location techniques will not identify the entire extent of a concept in the source code. In all instances, it simply guided the user to a part of the implementation of the concept. Impact analysis, change implementation and propagation finally revealed the full extent of the concept in the source code.

As mentioned in the introduction, the purpose of the concept location techniques is to reduce the space that the programmer has to search in order to find the concept. On this issue, all three concept location techniques were very effective, narrowing the space searched by the programmer by approximately three orders of magnitude.

Several facts were observed with respect to each technique, which allowed us to summarize their strengths and weaknesses, respectively.

The grep-based technique is very much dependent on the developer's existing knowledge about the system and the problem domain. If the user knows specific facts about the software, for example the default zooming value in the AOI case study, a good query can be easily written. Developers that are new to the system will have a hard time writing good queries. Potentially many queries will be needed before the result set is narrowed significantly.

The static dependency search is the most structured technique among these three and it can be used without any specific tool support since the dependencies can be followed manually through the code. Tool support is of course desirable for large systems. The technique depends on the programmer correctly understanding composite and local functionality of the classes. Every mistake in that leads to potentially costly backtracks in the search.

The IR-based technique suffers from similar problem as the grep-based technique; in order to formulate a good query knowledge about the software is needed, although more flexible queries are used here. In addition, the only way for the user to assess the quality of a query is to go through the list of results. On the other hand, since the technique returns a ranked list of results, the user has the potential to learn relevant information faster than with grep. Tool support is indispensable for this technique.

4.5. Threats to Validity

Several issues affected out results and limit our interpretations.

Doxygen is implemented in C++ and does not have a pure OO structure. There are many global functions and data types that are not encapsulated into classes. The granularity of the dependency search method had to be modified to deal with such functions, rather than classes. It is clear that in such systems there will be many concepts that are not cleanly encapsulated into one class.

The quantitative results (in our case LOC) are to a certain degree subjective, since they depend on the personal skills of the software engineer. Each of the concept location methods points the user to a part of the source code (i.e., a class, a method, or a line of text). It is up to the user how much code to investigate in the surrounding area to determine whether it is part of the concept or not. The three methodologies need to be further expanded and refined to address this issue.

Query formulations also depend on the personal ability of the users.

The investigated systems are by no means representatives for all types of OO systems. Generalization of some of our conclusions to all types of OO systems has to be done cautiously.

5. Conclusions and Future Work

The paper illustrates that concept location is an important programming activity even in OO programs. It also shows that in our cases, OO structuring does not offer any noticeable advantage for concept location.

The results of our case study complement conclusions in [29] and indicate that the individual concept location techniques display different strengths and weaknesses. We are investigating combinations of these techniques that would leverage their respective strengths, allowing the user to choose the most appropriate technique based on the circumstances of the project.

Part of that investigation is also a study on how concepts are represented in OO programs. Although select concepts are directly implemented as classes and are a highly visible part of the program architecture, other concepts are distributed through the system. Some of these issues are being addressed in the research on aspect oriented programming, but our primary research goal is the evolution and maintenance of existing OO systems and that may produce a different perspective on the issue.

Additional issues to be addressed in future work include investigation of concept location in multi-tier architectures that are becoming very common and are likely to pose new challenges to concept location.

6. 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 European Conference on Software Maintenance and Reengineering*, 2000, pp. 227-230.

[2] Biggerstaff, T. J., Mitbander, B. G., and Webster, D., "The concept assignment problem in program understanding", in

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.

Proceedings International Conference on Software Engineering, Baltimore, Maryland, 1993, pp. 482 - 498.
[3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem", *Communications of the ACM*, 37, 5, 1994, pp. 72-82.
[4] Chen, K. and Rajlich, V., "RIPPLES: Tool for Change in Legacy Software", in *Proceedings International Conference on Software Maintenance*, 2001, pp. 230 - 239.
[5] CygWin, "CygWin", Date Accessed: 09/30/2004, Online at http://cygwin.com, 2004.
[6] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, 41, 1990, pp. 391-407.
[7] Devanbu, P. T., "GENOA - a customizable, front-end-retargetable source code analysis framework", *ACM Transactions on Software Engineering and Methodology*, 8, 2, 1999, pp. 177-212.
[8] Eastman, P., "Art of Illusion", Date Accessed: 10/08/2004, Online at http://aoi.sourceforge.net/, 2004.
[9] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, 29, 3, 2003, pp. 210 - 224.
[10] Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E., "A Cliche'-Based Environment to Support Architectural Reverse Engineering", in *Proceedings International Conference on Software Maintenance*, 1996, pp. 319-328.
[11] Gold, N., "Hypothesis-Based Concept Assignment to Support Software Maintenance", in *Proceedings International Conference on Software Maintenance*, 2001, pp. 545-548.
[12] Harman, M., Gold, N., Hierons, R., and Binkley, D., "Code Extraction Algorithm which Unify Slicing and Concept Assignment", in *Proceedings Working Conference on Reverse Engineering*, 2002, pp. 11-20.
[13] Karakostas, V., "Intelligent search and acquisition of business knowledge from programs", *Journal of Software Maintenance: Research and Practice*, 4, 1, March 1992, pp. 1-17.
[14] Kiczales, G., Hilsdale, E., Hunguin, J., Kersten, M., Palm, J., and Griswold, W. G., "Getting started with ASPECTJ", *Communications of the ACM*, 44, 10, 2001, pp. 59-65.
[15] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J., "Aspect-Oriented Programming", in *Proceedings European Conference on Object-Oriented Programming*, 1997, pp. 220-242.
[16] Liu, H. and Lethbridge, T., "Intelligent Search Techniques for Large Software Systems", in *Proceedings IBM Centre for Advanced Studies and the National Research Council of Canada Conference*, 2001, pp. 10-18.
[17] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, 17, 8, 1991, pp. 800-813.
[18] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in *Proceedings IEEE Working Conference on Reverse Engineering*, 2004, pp. 214-223.
[19] Michail, A., "Browsing and Searching Source Code of Applications written using a GUI Framework", in *Proceedings International Conference on Software Engineering*, 2002, pp. 327-337.
[20] Müller, H. A., Tilley, S. R., Orgun, M. A., Corrie, B. D., and Madhavji, N. H., "A reverse engineering environment based on spatial and visual software interconnection models", in *Proceedings ACM SIGSOFT Symposium on Software Development Environments*, 1992, pp. 88-98.
[21] Murphy, G. C. and Notkin, D., "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, 5, 3, July 1996, pp. 262-292.
[22] Paul, S. and Prakash, A., "A Framework for Source Code Search Using Pogram Patterns ", *IEEE Transactions on Software Engineering*, 20, 6, 1994, pp. 463-475.
[23] Paul, S., Prakash, A., Buss, E., and Henshaw, J., "Theories and techniques of program understanding", in *Proceedings conference of the Centre for Advanced Studies on Collaborative research*, 1991, pp. 37-53.
[24] Rajlich, V., "A Methodology for Incremental Change", in *Extreme Programming Perspectives*, Marchesi, M., Succi, G., Wells, D., and Williams, L., Eds., Addison Wesley, 2002, pp. 201-214.
[25] Robillard, M. P. and Murphy, G. C., "FEAT a tool for locating, describing, and analyzing concerns in source code", in *Proceedings International Conference on Software Engineering*, Portland, OR, 2003, pp. 822-823.
[26] Salton, G. and McGill, M., *Introduction to Modern Information Retrival*, McGraw-Hill, 1983.
[27] Singer, J., Lethbridge, T., Vinson, N., and Acquetil, N., "An Examination of Software Engineering Work Practices", in *Proceedings conference of Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, November 10-13 1997, p. 21.
[28] van Heesch, D., "Doxygen", Date Accessed: 12/01/2004, Online at http://www.stack.nl/~dimitri/doxygen/, 2005.
[29] 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*, 65, 2, 2003, pp. 105-114.
[30] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, 7, 1, 1995, pp. 49-62.
[31] Wong, E., Gokhale, W., S., S., and Horgan, J. R., "Quantifying the closeness between program components and features", *Journal of Systems and Software*, 54, 2, 2000, pp. 87-98.
[32] Yin, R. K., *Case Study Research: Design and Methods*, 2nd ed., Sage Publications, 1994.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 24, 2008 at 15:50 from IEEE Xplore. Restrictions apply.