Towards Recovering Architectural Concepts Using Latent Semantic Indexing*

Pieter van der Spek, Steven Klusener
VU University Amsterdam
Amsterdam, The Netherlands
{pvdspek,steven}@cs.vu.nl

Pierre van de Laar
Embedded Systems Institute
Eindhoven, The Netherlands
pierre.van.de.laar@esi.nl

Abstract

Software engineers think about an existing software system in terms of high-level models. The high-level models are translated to source code and the concepts represented in these models are spread out over various parts of the software system. We propose to address the problem of locating these concepts using an advanced information retrieval method to exploit linguistic information found in source code, such as variable names and comments. Our technique is based on Latent Semantic Indexing (LSI). Applying LSI to source code, however, is not straightforward. Our approach therefore not only includes LSI, but also several other algorithms and methods. We will discuss each of these options and provide an overview of their effects using the results obtained from a case study.

1 Introduction

One of the major challenges in software maintenance is to determine the effects of modifications made to a program [10]. The overall cost of a small change (affecting only a handful of lines of code) can already be extremely high. By not just looking at the syntax of the source code, but rather at its meaning, we want to uncover what is being implemented. As the names of methods and variables are based on the concepts being implemented [15], it is possible to locate the architectural concepts embedded in the source code even when the concept is located in different, independent layers of the software system. Being able to locate the concepts which need maintenance more precisely, reduces the time needed to make a modification and aids software engineers in understanding the software system.

Our approach to recover architectural concepts embedded in the source code is based on Latent Semantic Indexing (LSI) [2, 9]. LSI is a technique in natural language processing, in particular in vectorial semantics, for analyzing relationships between a set of contexts[1] and the terms they contain by producing a set of concepts related to the contexts and terms. On the Web, LSI has been used by search engines as an auxiliary technique. An example of this is Yahoo! pay-for-performance paid collection [5].

Applying LSI to source code is not as straightforward as applying it to documents written in a natural language. Next to LSI, we have therefore included a set of algorithms and methods to prepare the source code before LSI is applied. Most of these algorithms and methods have been applied before. Our contribution is that we applied them in a structured manner and, more importantly, we examined their actual usefulness to the entire approach.

We then apply LSI to recover the architectural concepts by clustering variable names based on their similarity as opposed to clustering contexts as a whole. Whereas clustering only contexts shows the most important concept present in a context, clustering variable names can show multiple concepts occurring in a context and thus provide a finer level of granularity. Variable names that are clustered together represent a single architectural concept.

Structure of the paper. The remainder of this paper is structured as follows. In Section 2 we provide an overview of related work. Section 3 presents an overview of our approach for concept location using LSI. The results of a case study are presented in Section 4. Finally, Section 5 contains the conclusions and presents future work.

2 Related Work

The work presented in this paper addresses two specific issues: the location of concepts and features in the source code, and the use of information retrieval methods to support software engineering tasks and activities.

*This work has been carried out as a part of the DARWIN project at Philips Medical Systems under the responsibilities of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

[1] We use the word context as opposed to document, which is usually used in LSI. This avoids confusion between document as used in LSI terminology and regular documents.

Traditional concept location techniques can be divided into static approaches which use constructs provided by the programming language like *include*- and *inherit*-relations [1] and dynamic approaches using information gathered at runtime [4, 6]. Both approaches suffer from several drawbacks. The static analysis can only show relations if they are hard coded into the sources of the software system. With dynamic analysis the completeness of the results is very much dependent on the scenarios (use cases) and even then some elements of a concept might only show themselves under certain conditions.

Recently, information retrieval methods and specifically LSI have been used for locating concepts in source code [8, 12, 13]. Although our approach is similar to some of the existing methods as similar steps are used to preprocess the source code, these existing approaches do not use the variable names to identify concepts. Instead they use the term-by-context matrix to relate contexts, whereas we will relate terms, which we believe to provide more specific concepts.

3 Concept Location using LSI

In order to locate the conceptual clusters our approach takes several steps to prepare the source code and acquire the necessary information before LSI can actually be applied. Currently, we limit ourselves to source code only. However, documentation on the software system could also be used to enrich the context or to identify links between documentation and source code [11].

The remainder of this section provides an overview of our approach and shortly discusses each of the individual steps. Our prototype implementation of the approach allows for executing each of these steps separately using a variety of settings. It has been implemented by combining existing components which have been linked together using Perl and shell scripts.

3.1 From Variable Names to Term-by-Context Matrix

LSI takes contexts and terms as input. As LSI originates from natural language processing, applying it to source code is not straightforward. When dealing with a heterogeneous software system which uses various implementation technologies, the choice for contexts is not immediately obvious. As methods are common for most programming languages and generally implement a single, well-defined task, we have chosen them as are basic contexts. Comments preceding as well as those inside the methods can also be used to enrich the context.

3.1.1 From Variable Names to Terms

Variable names are most likely the best choice for using as terms, but their vocabulary is not as well-defined as is the case with for instance English words. Their meaning is obscured by abbreviations and various word combinations. Therefore, extracting variable names from the source code and using them unmodified as input for LSI most likely will not provide us with a good result.

We have tried several methods and algorithms to compensate for these problems. We will shortly discuss the most successful methods below. Figure 1 also lists the other methods.

Filtering. In order to reduce noise introduced by the occurrence of common variable names, e.g. *i*, *j*, *temp*, as well as common stop words occurring in comments we have manually selected only those variable names which capture some of the domain knowledge implemented in the software system.

Splitting. Variable names often consist of multiple words separated by an underscore ('_'), a hyphen ('-'), capitals (CamelCase), or numerical digits ([0-9]). This information can be used to split up variable names which makes it possible to identify relations using common words even though the actual variable names differ. An additional choice was to retain the original variable name next to the words as two different variable names can consist of the same words (e.g. *slice_to_pat* and *pat_to_slice*).

3.1.2 From Terms to Term-by-Context Matrix

Using the list of terms and the set of contexts, we count the number of times each term occurs within a specific context. This results in the term-by-context matrix whose $[i, j]^{th}$ element indicates the association between the i^{th} term and the j^{th} context.

Weighting. In order to balance out the influence of very rare and very common terms, we apply a term-weighting approach to the term-by-context matrix. Commonly, a simple document scoring method using relative term frequency (tf) and inverse document frequency (idf) [3] is used. The term-by-context matrix that results from these actions serves as input the LSI process which is described in the next paragraph.

3.2 Applying Latent Semantic Indexing

Using LSI an approximation of rank *k* of the original term-by-context matrix is created. In this way, anecdotal

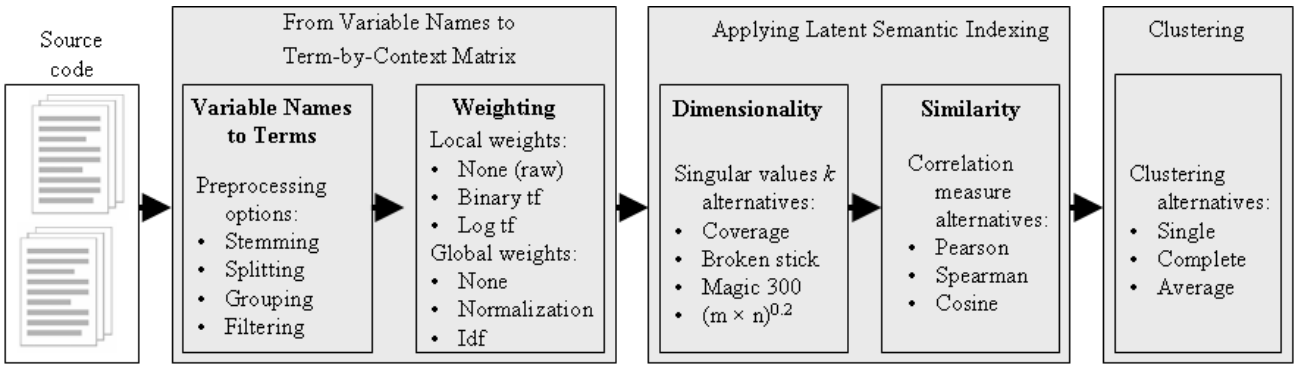

Figure 1. From source code to architectural concepts.

instances of terms are eliminated. Also, the original matrix lists only the words actually occurring in each context, whereas we are interested in all words related to each context – generally a much larger set due to synonymy.

The quality of the results is very much dependent on the choice for the value of *k*. Although most researchers use a fixed (or magic) value for *k*, a recent study by Kuhn et al. [8], in which LSI was applied to source code, describes an algorithm with which *k* can automatically be calculated. For an $m \times n$-dimensional matrix: $k = (m \times n)^{0.2}$. Further studies on the choice of *k* are open for future work.

To compute the similarity values, an often used measure is the dot product or cosine between the vectors. Although several alternatives exist, the cosine similarity measure tends to work well [9]. The matrix containing the similarity values is used to obtain the conceptual clusters which we are looking for.

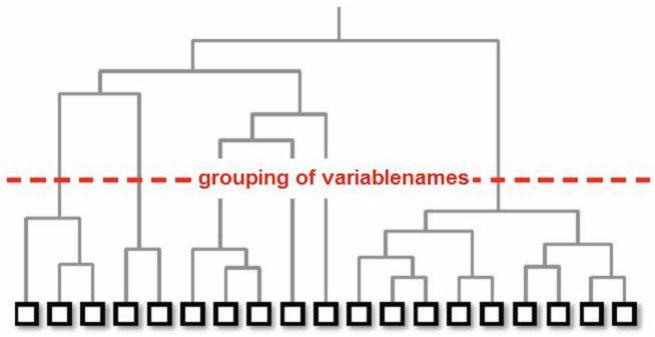

Figure 2. Creating clusters using hierarchical clustering.

3.3 Identifying Concepts in Source Code

Using the similarity measure it is possible to cluster the related terms using a variety of clustering algorithms. For our case study we have used complete-link hierarchical clustering Figure 2. In the complete-link algorithm, the distance between two clusters is the maximum of all pairwise distances between patterns in the two clusters.

As we do not know how many clusters exist, a partitioning algorithm is not usable as such an algorithm requires the desired numbered of clusters as input value. Hierarchical clustering, on the other hand, does not have this requirement. Furthermore, the complete-link algorithm produces tightly bound or compact clusters and does not suffer from a chaining effect [7].

4 Case Study

As has been explained in paragraph 3.1 there are a lot of ways in which the input can be preprocessed. Each of the measures influences the outcome of the analysis to a certain extent. Also, combining multiple measures can both enhance and negate these effects. In this paragraph we want to discuss the results from one of the experiments we conducted. For the case study we have used the source code from the magnetic resonance imaging scanners of Philips Healthcare. We have chosen a subset of the software system on which we have applied our approach as this made the results easier to verify. Whereas the entire software archive consists of about 7 million lines of code mostly consisting of C, C++, and C#, the subset we have chosen consisted of 7 different components consisting of 169 files containing around 190.000 lines of code.

4.1 From Variable Names to Terms

In order to move from variable names to terms, we have considered several options as described in Section 3.1. For the remainder of this section we will focus on creating terms from variable names using the following two steps: filtering, and splitting. We will also show the effects of weighting applied to the list of terms.

Filtering. Filtering the list of variable names can be done in two ways: (1) excluding meaningless variable names based on a set of criteria or patterns or (2) only including meaningful variable names which bare meaning (based on expert or domain knowledge). Of these two, the latter is more restrictive.

In our tests, the more restrictive approach improved the results dramatically. Whereas without adding this domain knowledge our approach could only identify a single, generic concept, with the domain knowledge, however, we were able to identify several concepts which were roughly similar to what was expected by the experts.

A downside to this approach is that it requires the input of experts. This can cause the outcome to reflect what they already know. A better solution would be to determine important variable names automatically based on information from the source code and possibly even documentation. This is left for future research.

Table 1. Similarity values for two variable names without and with splitting applied.

Most similar to:	Without splitting	With splitting
ctry_code	*lang_name*: 1.0	*lang_name*: 1.0
lang_ctry_code	*logfilespec*: 1.0	*ctry_code*: 0.91

Splitting. Splitting up variable names into their constituent parts should help to identify relations between similar variable names (i.e. variable names having one or more words in common). In practice, this effect can also be observed. Very often variable names related to a single concept do not occur together in the same context. Using the constituent parts of variable names, however, it is possible to link them as they do contain similar parts. Table 1 shows an example of this effect.

4.2 From Terms to Term-by-Context matrix

The effect weighting has on the similarity values can vary greatly and is not easy to predict as the matrix approximation, as performed by the LSI algorithm, has a profound influence on the resulting values. An example is shown in Table 2.

Table 2 shows the effect, or lack thereof, weighting has on the similarity values. In both cases we would expect a high similarity with *memory*. Without applying weighting, however, neither variable names score very high. In fact, the score is 0, meaning there is no similarity whatsoever. Using the tf-idf weighting scheme, this view changes, but only for one of the two variable names showing that, although weighting can have a positive effect, the effect is not consistent and is hard to predict. In general, however, the results are an improvement over the case in which weighting is not applied.

4.3 Applying Latent Semantic Indexing

After the construction of the term-by-context matrix, LSI finds a low-rank approximation to the original term-by-context matrix. We started the experiment with 4020 variable names. This initial set of variable names provided us with 652 terms occurring in 1360 contexts such that $k = (652 \times 1360)^{0.2} \approx 15$. Table 3 shows a small selection of examples. The first row shows how the truncated matrix contains a term-by-context value even though the original matrix contains zeroes, thus compensating for synonymy. The opposite effect is shown in row 2, thus compensating for anecdotal instances of terms. The remaining rows in Table 3 show how values can both stay the same as well as change to another non-zero value in the truncated matrix. The large differences in the first two rows can be explained by looking at the eigenvectors for these terms. Where the eigenvalues of *interactorInfo* corresponding to the 15 most significant singular values all non-zero, some of the eigenvalues of *option_set_arr* are close to zero resulting in a low value in the low-rank approximation of the term-by-context matrix.

Table 3. Term-by-context values.

Term	Context (methods)	Orig.	Trun.
1. *interactorInfo*	ActivateInteractor	0.0	8.26
2. *option_set_arr*	initialize_optionsetp	30.34	0.001
3. *norm*	optimize_sign_-and_inplane	25.86	1.77
4. *coil*	AWCOIL_synch	23.51	23.51

Table 4. Term-Term similarity values.

Term	Term	Orig.	Trun.
1. *Override-Minimum-Annotations*	*OverrideFull-Annotations*	1.0	1.0
2. *iconLayoutColumn-Remove*	*textLayoutAdd-Column*	1.0	0.0
3. *coil_descriptor*	*grcoil_ptr*	0.0	1.0

Finally, Table 4 shows how the similarity measures change between the original and the truncated term-by-context matrix. By truncating the original term-by-context matrix similarity values sometimes do not change at all (e.g. row 1), but they can also change quite drastically (e.g. row

Table 2. Similarity values for two variable names without and with weighting applied.

	Variable name:	
	requestMemoryAmountPrServer	***baseMemoryPRserver***
Term frequency	Context1 - 4.0 Context2 - 3.0	Context3 - 1.0 Context4 - 1.0
Term frequency (weighted)	Context1 - 11.66 Context2 - 10.04	Context3 - 4.54 Context4 - 4.54
Similarity to *memory*	0.0	0.0
Similarity to *memory* (weighted)	0.01	1.0

2 and 3). This behavior has been examined more closely in [14].

4.4 Identifying Concepts in Source Code

The similarity measures serve as input to the final clustering step. Using complete-link hierarchical clustering we construct a dendrogram which can than be cut at a certain similarity level. The exact level at which the tree should be cut can vary, but we found 0.85 to provide accurate results. Due to the steps taken to compile a list of terms from the variable names found in the source code, the precision was already quite high and thus a high similarity level can be chosen. This results in compact clusters with a high internal similarity.

We have evaluated these results together with experts at Philips Healthcare. From these discussions it became clear that the level of abstraction of the concepts identified in this way tends to vary and is not consistent throughout the entire set of results.

Furthermore, these concepts are sometimes located within a single building block or even a single source file, whereas other concepts are spread out over various building blocks. Especially the latter type of concepts were considered interesting as these can cause problems when the software system has to be modified. The fact that these kinds of concepts are identified and validated shows the potential of our approach.

5 Conclusions

From the discussion above it has become clear that LSI has a lot of promise when it comes to recovering architectural information from source code. Unfortunately, the quality of the outcome can vary greatly using different methods for preprocessing the input. Both splitting of variable names and carefully selecting variable names resulted in large improvements in performance. Applying a weighting scheme helped to further reduce the influence of very rare and common terms.

Still work remains when it comes to properly selecting the variable names, comparing the results to concepts identified by experts and retrieving more consistent concepts with respect to the level of abstraction. Furthermore, we would like to include structural information in order to raise better results.

Based on our experiments we are confident that LSI is able to identify architectural concepts in source code. Identifying these concepts by clustering the variable names instead of clustering documents results in more precise clusters as documents can contain multiple concepts. This shows that LSI can be a powerful tool to assist in supporting many activities of the software maintenance process.

References

[1] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the IWPC*, pages 241–247, 2000.

[2] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[3] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(3):229–236, 1991.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Trans. on Soft. Eng.*, 29(3):210–224, 2003.

[5] D. Gleich and L. Zhukov. SVD based term suggestion and ranking system. In *Proceedings of the ICDM*, pages 391–394, Washington, DC, USA, 2004. IEEE Computer Society.

[6] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the CSMR*, 2005.

[7] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[8] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, 2007.

[9] T. K. Landauer, P. W. Foltz, and D. Laham. Introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[10] J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the ICSM*, pages 282–291, 1993.

[11] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the ICSE*, pages 125–135, 2003.

[12] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the ICSM*, pages 133–142, 2005.

[13] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the IWPC*, pages 33– 42, 2005.

[14] R. Newo Kenmogne. *Understanding LSI Via The Truncated Term-Term Matrix*. PhD thesis, Universität des Saarlandes, 2005.

[15] J. Sajaniemi and R. N. Prieto. An investigation into professional programmers' mental representations of variables. In *Proceedings of the IWPC*, pages 55–64, 2005.