Identification of High-Level Concept Clones in Source Code

Andrian Marcus, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
amarcus@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

Source code duplication occurs frequently within large software systems. Pieces of source code, functions, and data types are often duplicated in part, or in whole, for a variety of reasons. Programmers may simply be reusing a piece of code via copy and paste or they may be "re-inventing the wheel".

Previous research on the detection of clones is mainly focused on identifying pieces of code with similar (or nearly similar) structure. Our approach is to examine the source code text (comments and identifiers) and identify implementations of similar high-level concepts (e.g., abstract data types). The approach uses an information retrieval technique (i.e., latent semantic indexing) to statically analyze the software system and determine semantic similarities between source code documents (i.e., functions, files, or code segments). These similarity measures are used to drive the clone detection process.

The intention of our approach is to enhance and augment existing clone detection methods that are based on structural analysis. This synergistic use of methods will improve the quality of clone detection. A set of experiments is presented that demonstrate the usage of semantic similarity measure to identify clones within a version of NCSA Mosaic.

1. Introduction

Research suggests [3, 23] that a reasonable amount of large software systems contain duplicated implementations of source code. There are a number of reasons for the existence of these duplicate implementations, or *clones*.

For one, programmers often perform a type of ad hoc reuse by using the copy and paste method. The scenario is common; you find a piece of code in another routine that almost solves your problem. You copy it to your routine and modify it to suit the problem at hand. This type of "reuse" is less costly (at the time) than redesigning a larger part of the system to incorporate the necessary generality of the reused piece of code. Ideally, the program would create a more general set of routines or design a class hierarchy to solve the reusing problem. This represents a programmer's *explicit* intent to reuse an abstraction in the problem, or solution, domain. Baxter et al. go as so far to say that we should offer tools to support this type of cloning (reuse) in a more structured and well-defined manner.

The above described situation gives rise to the following types of clones. A (perfect) clone is a program fragment that is identical to another program fragment. A *near miss clone* is a program fragment that is very similar to another fragment. The near miss clone comes about when the programmer modifies the copied fragment.

Another reason for the occurrence of clones, especially in very large software systems, is because of "re-inventing the wheel". A developer (or maintainer) may not know of the existence of a solution to their problem and they just solve it by developing new code. Alternatively, they may know of a fragment that is similar to what they need, but feel the expense of understanding and modifying the fragment is to great in comparison to "writing it themselves". Re-inventing the wheel gives rise to near miss clone and possibly *"wide miss" clones*. A wide miss clone solves the same (or nearly the same) problem but has a very different structure. While this general problem could be solved by better designs, communication among developers, or better documentation, it remains a reality.

From our experience, these types of clones often manifest themselves as higher-level abstractions in the problem or solution domain. A simple example that comes to mind is an ADT list. A list structure is often duplicated in one form or another throughout a system. Each programmer, or team, builds one to suit his or her particular needs.

We term these types of clones as *high-level concept clones*. While a number of the existing clone detection methods can detect some of these types of clones, no

method directly addresses the identification of high-level concept clones.

In this paper, we present a method that addresses the detection of high-level concept clones. We feel that this method should be used in conjunction with other existing methods to synergistically identify all types of clones. The next section describes the underlying approach to our method. Section 3 presents how we detect high-level concept clones. Section 4 describes the results of applying this approach to a medium sized software system (Mosaic). Section 5 reviews the related work and the remaining sections describe our current directions.

2. Detection of high-level concepts in code

The method we use to detect high-level concepts in source code is derived from our work on the PROCSSI system [28]. PROCSSI is composed of a set of methods to support software maintenance tasks. The main feature of PROCSSI is the use of both semantic and structural information extracted from the source code. Here we use the term semantic to refer to the information embedded in the source that links it to the problem or solution domain. For example, the term "sort" has a clear meaning to a programmer and its presence in a piece of source code typically has great significance with respect to understanding of the source. Structural information refers directly to concepts such as the structural organization, control flow, and data flow of the source code. A large number of (inexpensive) methods to extract structural information exist but few methods, beyond an hourly wage, exist to map source code to the problem and solution domain.

For the semantic dimension, PROCSSI uses the profiles generated by information retrieval methods, in this case a vector representation from Latent Semantic Indexing (LSI), to compare components and classify them into clusters of semantically similar concepts (the details of LSI are presented later in this section). PROCSSI works as follows. Given a software system, it is broken down into a set of individual source code documents. A simple parsing of the source code is done to break the source into the proper granularity and remove any non-essential symbols and text. Comment delimiters and many syntactical tokens can be removed as they add little or no knowledge of the problem domain. The profile of each source code document generated by LSI can then be used to cluster the documents in to related groups. Clustering of source code based on semantic and structural information is very useful in the maintenance and evolution of legacy software systems. For instance, the clustering can be used to assist in the re-modularization [30, 34, 35, 39] of systems and the identification of abstract data types [8, 17]. If a software system were to be reengineered into an object-oriented language from a structured one, then this type of clustering would prove to be very useful. The objective is to reduce the amount of source code an engineer needs to view, at any one time, and give them clues about possible relationships within the system not apparent from the current organization of the files or documentation.

PROCSSI uses a simple graph theoretic approach for clustering, but a number of other types of clustering algorithms have been used to cluster software [1, 2, 18, 25]. Details of our approach on how the high-level concepts are identified using this clustering method can be found in [28].

2.1. Information retrieval and software

There are a variety of information retrieval methods including traditional [11, 37] approaches such as signature files, inversion, and clustering. Other methods that try to capture more information about documents to achieve better performance include those using parsing, syntactic information, natural language processing techniques, methods using neural networks, and advanced statistical methods. Much of this work deals with natural language text and a large number of techniques exist for indexing, classifying, and retrieving text documents. These methods produce for each document a profile. A profile is an abbreviated description of the original document that is easier to manipulate.

The research that has been conducted on the specific use of applying information retrieval methods to source code and associated documentation typically relates to indexing reusable components [13-15, 25, 26, 29, 32]. Notable is the work of Maarek [25, 26] on the use of an IR approach for automatically constructing software libraries. The success of this work along with the inefficiencies and high costs of constructing the knowledge base associated with natural language parsing approaches to this problem [10] are main motivations behind our research. In short, it is very expensive (and often impractical) to construct the knowledge base(s) necessary for parsing approaches to extract even reasonable semantic information from source code and associated documentation. Using IR methods (based on statistical and heuristic methods) may not produce as good of results, but they are inexpensive to apply and coupled with the structural information of the program, should produce good quality and low cost results.

2.2. Latent semantic indexing

Latent Semantic Indexing (LSI) [6, 24] is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of natural

language) reflective in their usage. The method generates a real valued vector description for documents of text. This representation can be used to compare and index documents using a variety of similarity measures. We apply LSI to source code and its associated internal documentation (i.e., comments) and then use the similarity measures to induce the similarity of different source code documents.

Work applying LSI to natural language text by [6, 24] has shown that that LSI not only captures significant portions of the meaning of individual words but also of whole passages such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about word contexts in which a particular word appears, or does not appear, provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other.

One of the criticisms of this method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. But very good representations and results are derived without this information [7]. This characteristic is very well suited to the domain of source code and internal documentation. Because much of the informal abstraction of the problem concept may be embodied in names of key operators and operands of the implementation, word ordering has little meaning. Source code is hardly English prose, but through the use of selective naming, much of the high level meaning of the problem-at-hand is conveyed to the reader (programmer/developer). Internal source code documentation is also commonly written in a subset of English [10] that may also lend itself to the IR methods utilized.

Like some other IR methods LSI does not utilize a grammar or a predefined vocabulary. Though, many IR methods do use a list of non-essential words with low discriminatory power. This makes automation much simpler and supports programmer defined variable names that have implied meanings (e.g., avg) yet are not in the English language vocabulary. The meanings are derived from usage rather than a predefined dictionary. This is a stated advantage over using a traditional natural language approach, such as in [10], where a (subset) grammar for the English language must be developed.

3. Identifying high-level concept clones

The method we propose for identifying high-level concept clones is based on the semantic similarity measure between source code documents described earlier. These similarity measures are akin to the work by [16, 22, 38]. They compute the similarity between software elements based on structural information, with the purpose of identifying high-level concepts in code.

In its current form, the approach is an automated assistant to the developer in the identification of clones. To fully automate the process there is a need for integration with other clone detection methods that are based on structural information. Section 6 of this paper will discuss this issue.

Once the semantic similarities between the source-code documents are computed, the user should select a group of documents that implement a known high-level concept (e.g., and ADT) as starting point. There are several ways in which a user, even without much knowledge of the given software at hand, can identify the implementation of such a high-level concept. Either by selecting groups of documents based on file names and/or function names, or by using a clustering of the source code documents, such as described in [27, 28]. Automation of this part is also possible, by using selected metrics as guidance. In previous work [28], we used the semantic similarity measures to assess the semantic cohesion of clusters of source code documents. One such cluster could be determined using structural information about the code such as control or data dependency. Such clusters of source code documents could be a cross multiple files. Most of the existing clone detection methods try to identify clones within or between files. This is the approach we consider as well. Some variation of the measures and metrics proposed in [28] are defined here with respect to clone detection.

Definition. A *source code document* (or simply document) d is any contiguous set of lines of source code and/or text. Typically, a document is a function, block of declarations, definitions, or a class declaration including its associated internal documentation (comments).

Definition. A *software system* is a set of documents $S = \{d_1, d_2, \ldots, d_n\}$. The Total number of documents in the system is $n = |S|$.

Definition. A *cluster*, c_k, is a set of documents from S such that $c_k \subseteq S$. Size of a cluster, c_k, is the number of documents in a cluster, noted $|c_k|$.

Definition. A *file* f_i, is composed of a number of contiguous documents and the union of all files is S. Size of a file, f_i, is the number of documents in the file, noted $|f_i|$.

Definition. The software system is represented as a *relationship graph* that is a multi-graph $G = (S, E)$, where the nodes S are the documents, E is a set of weighted edges, and a relation e: $E \rightarrow \{(d_i, d_j) \mid d_i, d_j \in S; d_i \neq d_j\}$. The relation e defines which nodes are connected by which edge. The edges u and v are called parallel or multiple edges if $e(u) = e(v)$.

Each parallel edge represents a relationship between the nodes (i.e., source code documents) it connects. There could be different types of edges; each represents different relationships between two source code documents. Here

we consider two types of relationships, namely semantic similarity and structural relationships. The structural relationship is defined by the file structure of the system.

Using the semantic similarity measure one could cluster the software system using a variety of clustering algorithms. Once such a clustering is performed, the following measures and metrics will be computed.

Definition. The number of clusters that contain a document from a given file is $|CDF_i|$ where

$$CDF_i = \{c_k \subseteq S \mid c_k \cap f_i \neq \emptyset\}.$$

Definition. The semantic cohesion of a file with respect to clusters is

$$SCFC_i = 1 - \frac{|CDF_i| - 1}{|f_i|}.$$

Definition. Number of files related by a cluster to a given file, f_i, is $|RF_i|$ where

$$RF_i = \{f \subseteq S \mid c_k \cap f \cap f_i \neq \emptyset, f \neq f_i, c_k \subseteq S\}.$$

Definition. Number of files strongly related by a cluster to a given file, f_i, is SRF_i: $SRF_i = |RF_i| - \max |c_k| - 1$ and $c_k \in LC_k$ where LC_k is the set of clusters that contain documents from f_i and have a low semantic cohesion with respect to files.

$$LC_k = FDC \cap \{c_j \subseteq S \mid 1 - \frac{|FDC_j| - 1}{|c_j|} < \varepsilon\}$$

where ε is an empirically established threshold.

In addition we consider two files f_i and f_j as being related if there is at least one document is f_i that is similar with at least one document in f_j:

$$\exists\, d_i \in f_i, d_j \in f_j \text{ such that } sem(d_i, d_j) > \alpha,$$

where *sem* is the semantic similarity function (i.e., the cosine between the vector representation if the two documents), and α is a threshold (0.7 in this case that corresponds to a 45 degree angle).

Files with high semantic cohesion are of interest with respect to clone detection. Such files most likely contain implementation of only one or very few high-level concepts and versions of them. Among these files, we need to focus on those that are strongly related to other files. All these computations are automated and a prototype tool has been developed and tested previously, using a minimal spanning tree algorithm for clustering. Other clustering algorithms are under investigation, which may provide better results for clone detection.

At this point in the process, user intervention is necessary. As we specified earlier, the high-level concept we identify, and for which we are trying to find clones, have a rather imprecise definition. This definition is based on the user's understanding of the system and represents some mapping between the source code and the problem or the solution domain. Further automation would require the acquisition and representation of extensive domain knowledge - an extremely expensive and difficult undertaking that is necessary for each system examined. Thus, human interaction is unavoidable at this point in the process.

Two types of clones are to be identified. One will consist of groups of documents from any file that implements the same high-level concept, while the two sets of documents have no data or control connections (e.g., separate implementation of the same or related ADT). The second type of clones will consist of groups of documents from different files that use the same data structure but implement their own set of operations. Obviously none of these clones will be exact ones.

Once a file (or group of documents) that contains the implementation of a high-level concept is chosen as starting point, the user will use the computed measures to identify clones. All the strongly related files have to be investigated by the user. While this may seem and extremely laborious task, experiments showed that the number of documents that need to be investigated is in fact reasonable and the guiding metrics drastically reduce the search space that a keyword-based search would generate. Matching these results with the ones provided by another clone detection method could further reduce this search space. This work is in progress and some details are provided in section 6. The following section describes a set of experiences we have done to show the usability of the method.

4. Experiments

A set of experiments was run to determine the suitability of our method for clone detection. The source code for an older version of Mosaic (v2.7) [33] was used as input into LSI and clustered using the previously described method.

Mosaic 2.7 is written in C and was programmed and developed by multiple individuals. No single coding standard is observed over the entire system and different standards are routinely used within a given file. Little or no external documentation on the design or architecture is available and the internal documentation is often scarce or missing. In short, Mosaic reflects the kinds of realities often found in commercial software due to the many external issues that affect a software development project.

Table 1 gives the size of the Mosaic system (269 files containing approximately 95 KLOC). A semantic space using a dimensionality of 350 for the 2,347 documents was

Table 1. Vitals for Mosaic.

LOC	95,000
Vocabulary	5,114
Number of parsed documents	2,347
Number of clusters produced	655

Table 2. A distribution of the size of clusters. The number of clusters that contain a given number of documents.

Number of Documents	Number of Clusters
1	481
2	98
3 - 5	46
6 - 10	15
11 -30	8
38	1
99	1
1084	1

generated. The documents were then clustered, as stated previously, based on the angle of 45 degrees or less (i.e., between 1.0 and 0.7) between any two vectors, which resulted in 655 groupings.

A distribution of the clusters based on the number of documents they contain is given in table 2. There are a large number of singleton clusters (481) and few really large clusters. These numbers reflect the same type of trends that were found in the earlier experiments. The large number of clusters of size one reflects the fact that many functions often stand by themselves semantically. The largest cluster is for the most part composed of a common header comment that is found in almost every file. It also includes a large number of very small documents that were parsed out to be only one or two lines of code.

Of interest here is the semantic cohesion of files and the number of related files to a given one. Table 3 summarizes the semantic cohesion values of the files based on the current clustering. Table 4 shows the distribution of documents within files.

The clusters of interest, with respect to clone detection, are those that have more than one but not excessive numbers of documents and the files with high semantic cohesion. Our assumption is that a high-level concept is implemented using more than a few functions and data types.

We chose to start by choosing a file with the desired properties that was identified as containing the implementation of a high-level concept. The file is `list.c`, which together with `list.h` and `listP.h` implement a linked list of character strings (`char*`). The file `list.c` has a semantic cohesion of 0.8 and contains 15 documents. A total of 19 documents are similar to at least one document in `list.c`, spanning over 11 files, determining 116 pair of similar documents. Out of the 19 similar documents, 10 documents are similar to at more 8 documents from `list.c`.

Table 3. Number of files with semantic cohesion within a given interval.

Number of Files	Semantic Cohesion
5	[0.9,1.0)
15	[0.8,0.9)
28	[0.7,0.8)
19	[0.6,0.7)
30	[0.5,0.6)
41	(0.0,0.5)
131	Contain 1 document

4.1. Results

Among the files that contain these documents, 2 were found to contain a different implementations of a linked list and 4 other files were implementing their own operations defining a linked list but using the same data structure defined in `list.h` or the data structure defined in `HTList.h`. The other files were either using the implemented list ADT in a correct manner or they turned out to be similar by accident (two cases).

Although the results are encouraging, due to the nature of the clustering algorithm used, the experiment was repeated twice more using as a starting point the pair of files that contain the different list implementations. These files are: `hotlist.h`, `hotlist.c`, `HTList.h`, and `HTList.c`. One could of course argue that the names of the files are self-explanatory and someone could have found these clones easily. If the files would have had a more cryptic name that did not hint to the word 'list' this action would not have been possible, while our method would work the same. In fact, the names of the files that contain different implementation of a list using the same data structure did not give any clues (e.g., `HTMLwidgets.c`, `cciBindings2.c`, etc.). Restarting the experiment yielded 4 more implementations of linked lists. Overall, 11 high-level concept clones were identified, implementing versions of a linked list.

A reengineering of the system would most likely be concerned with creating a single (more general) implementation of a linked list and reuse it through instantiation and/or inheritance rather than re-implementation.

Table 4. A distribution of the size of files. The number of files that contain a given number of documents.

Number of Documents	Number of Files
1	110
2 - 4	45
5 - 10	42
11 - 20	44
21 - 50	21
55 - 91	7

4.2. Comparison

Since a considerable part of the identification process is manual, our focus is on providing the user with best heuristics to which files and documents he or she needs to analyze. Measuring the overall effort is not easy in this case. In order to show the efficiency of the method we tried to identify the clones of the linked list ADT using

keyword search in the source code. For example, searching for the occurrences of the term 'list' yielded 3002 occurrences in 125 files. Searching for the occurrences of 'list' as a separate word yielded 838 occurrences in 82 files. Various other keyword searches using regular expressions and other terms yielded comparable results. With this in mind, our method not only helps identifying an implementation of a high-level concept in the first place, it also reduces the search space for clones by at least 5 times. This also supports other research's finding as to LSI's performing much better then simple word matching methods.

4.3. Limitations

In some cases, the developers of Mosaic choose to entirely rename the data structure and operation names in a cloning (re-implementation) of a list (e.g., a list of news records). When comments are also discarded, our measures were unable to detect similarities between two such implementations. Though this demonstrates the importance of internal documentation can have for source code understanding. By lowering the threshold for defining the similarity, we obtained too many false positives. Since the kind of clones we are attempting to identify typically contained groups of documents spanning over several functions and files, some of the documents will still contain features that could be identified by our method.

Still, limitations like this prompted us to experiment with combining this method with existing clone detection methods to increase the precision. Section 6 describes the experiments we are currently executing and our hypothesis on how much this will improve the clone detection process.

5. Related work

Existing research in clone detection is based on two major approaches: 1) using structural information about the code (e.g., metrics, AST, control/data flow, slices, structure of the code/expressions, etc.) [3, 5, 20, 21, 23, 31]; and 2) using string-based matches [3, 9, 19, 36]. Each of these methods has its advantages and disadvantages. The methods that fall in the first category are obviously language dependent, thus a bit less flexible, while some of the methods in the second category can only deal with exact matches and can have scalability problems due to the large number of comparisons needed.

Johnson [19] has developed a method for the identification of exact duplications of sub-strings in source code using fingerprints at file level granularity. Baker's tool, called DUP [3], finds exact matches and p-matches based on parameters (i.e., replacing identifiers). The granularity is that of chunks of source code larger than a given threshold (usually around 30 LOC). Comments are ignored in this string matching process.

A set of other related approaches use metrics derived from the structure or layout of the code to identify similarities between source code elements. Mayrand et al. [31] use a set of metrics to characterize functions based on name, layout, expressions, and control flow to identify duplicate, or near duplicate, functions in programs written in procedural programming languages. The method is also used by Lague et al. [23] in their attempt to integrate clone tracking into the development process. Kontogiannis [21] uses five complexity metrics as characteristics of code. The code segments are represented in this 5-dimensional space and Euclidian distance is used as similarity metric. The method is based on a system representation as an annotated syntax tree. This method resembles our approach in two ways, namely the source code segment is represented as a multi-dimensional space, and this representation is used to define a similarity measure.

On a different note, Ducasse et al. [9] propose a language independent method to identify clones. The method is based on simple string matching, textual reports, and scatter plot visualization. Comments are removed from the source code and the text preprocessing is based on a similar method to the Unix `diff`. The string-matching algorithm identifies exact matches only and the user identifies clones using the DUPLOC visualization tool [36].

Komondoor and Horwitz [20] use backward slicing on the program dependence graph to identify clones in C programs. The advantage of this method over the previous ones is that it can identify non-contiguous clones.

Complementing the research in clone detection are the clone removal methods. Clone detection, in general targets some aspects of software maintenance, trying to improve the quality of a software system under maintenance or development. Clone removal is aimed at supporting specific software engineering tasks namely, reengineering, reverse engineering, or program understanding. This paper is not concerned in detail about removal of the identified clones. Ideally, these high-level concept clones would be combined in one or two modules or classes during reengineering. Existing research describes several methods for clone removal [4, 5, 12].

6. Combining multiple detection methods

We are currently investigating the combination of methods through two different approaches. One approach is to apply two or more methods to the same code and then merge the results. Three methods that are based on structural information [5, 20, 31], using abstract syntax

tree representation, software complexity metrics, and slicing respectively, seem the best candidates to augment the results of our method. In addition, Baker's [3] p-matches could also be used to identify structurally similar code segments where variable names are completely changed.

Another approach is to augment the real-valued vector representation of the source code documents, produced by LSI, by adding more dimensions that would represent structural attributes of the source code derived from metrics. Metrics values such as those used in [21] or [31] seem most appropriate. Preliminary research on this approach seems promising and would enhance the descriptiveness of the LSI output to include structural type information.

7. Conclusions

It has been suggested in [5], that clone detection methods can be used to identify domain concepts in the code. Our method attempts to show that the opposite is true as well. This paper presents how high-level concept clones can be detected using a static analysis method that is designed for detection of domain concepts in the code. The granularity and type of clones identified through the proposed approach differ from definitions of clones in previous research. Here, we are looking for high-level concepts clones such as ADTs.

Researchers [5] recognize the need to detect "semantic equivalence" in source code, but due to practical reasons related to the difficulty and cost of this task, simpler equivalency definitions were used instead. Our approach identifies semantic similarities in source code but some lack of precision and limited automation is the price to pay for the low costs of the proposed method. None of the existing methods would be able to identify two different implementations of a function that inserts and element into a linked list that are not connected by data or control flow. Our method fails to identify two functions with similar structure and functionality if comments do not exist and the identifier names are completely different. The best way to overcome this aspect is by combining two or more clone detection methods and thus taking advantage of their respective strengths.

The paper shows that identifying clones based on semantic equivalence is possible at a relatively low cost and can be automated to a large degree by combining it with other clone detection methods. This also lays the grounds for future research that, as we stated earlier, is already in progress.

8. References

[1] Anquetil, N. and Lethbridge, T., "Extracting Concepts from File Names; a New File Clustering Criterion", in Proceedings of 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, 1998, pp. 84-93.

[2] Anquetil, N. and Lethbridge, T., "Experiments with Clustering as a Software Remodularization Method", in Proc. of 6th Working Conference on Reverse Engineering, 1999.

[3] Baker, B., "On Finding Duplication and Near-Duplication in Large Software Systems", in Proc. of Working Conference on Reverse Engineering, Toronto, Ontario, July 1995.

[4] Balazinska, M., Merlo, E., Dagenais, M., and Lague, B., "Partial Redesign of Java Software Systems Based on Clone Analysis", in Proceedings of Working Conference on Reverse Engineering, Atlanta, GA, October 6-8 1999, pp. 326-336.

[5] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L., "Clone detection using abstract syntax trees", in Proceedings of International Conference on Software Maintenance, Bethesda, Maryland, November 16-19 1998, pp. 368-377.

[6] Berry, M. W., "Large Scale Singular Value Computations", *International Journal of Supercomputer Applications*, vol. 6, 1992, pp. 13-49.

[7] Berry, M. W., Dumais, S. T., and O'Brien, G. W., "Using Linear Algebra for Intelligent Information Retrieval", *SIAM: Review*, vol. 37, no. 4, 1995, pp. 573-595.

[8] Canfora, G., Cimitile, A., Munro, M., and Tortorella, M., "Experiments in Identifying Reusable Abstract Data Types in Program Code", in Proceedings of IEEE 2nd Workshop on Program Comprehension, 1993, pp. 36-45.

[9] 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*, vol. 41, 1990, pp. 391-407.

[10] Ducasse, S., Rieger, M., and Demeyer, S., "A Language Independent Approach for Detecting Duplicated Code", in Proc. of International Conference on Software Maintenance, Oxford, England, Aug 30 - Sep 3 1999, pp. 109-118.

[11] Duda, R. O. and Hart, P. E., *Pattern Classification and Scene Analysis*, Wiley, 1973.

[12] Dumais, S. T., "Latent Semantic Indexing (LSI) and TREC-2", in Proceedings of The Second Text Retrieval Conference (TREC-2), March 1994, pp. 105-115.

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

[14] Faloutsos, C. and Oard, D. W., "A Survey of Information Retrieval and Filtering Methods", University of Maryland, Technical Report CS-TR-3514, August 1995.

[15] Fanta, R. and Rajlich, V., "Removing clones from the code", *Journal od Software Maintenance: Research and Practice*, vol. 11, no. 4, 1999, pp. 223-243.

[16] Fischer, B., "Specification-Based Browsing of Software Component Libraries", in Proc. of 13th ASE, 1998, pp. 74-83.

[17] Frakes, W., "Software Reuse Through Information Retrieval", in Proc. of 20th Annual HICSS, Kona, HI, Jan. 1987, pp. 530-535.

[18] Girard, J.-F., Koschke, R., and Schied, G., "Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding", in Proceedings of Working Conference on Reverse Engineering, 1997, pp. 66-75.

[19] Girard, J.-F., Koschke, R., and Schied, G., "A Metric-Based Approach to Detect Abstract Data Types and State Encapsluation", *Journal Automated Software Engineering*, vol. 6, no. 4, October 1999.

[20] Girard, J.-F. and R., K., "A Comparison of Abstract Data Type and Objects Recovery Techniques", *Journal Science of Computer Programming, Elsevier* 1999.

[21] Hutchens, D. and Basili, V., "System Structure Analysis: Clustering With Data Bindings", *IEEE Transactions on Software Engineering*, vol. 11, no. 8, 1985, pp. 749-757.

[22] Johnson, H. J., "Substring matching for clone detection and change tracking", in Proceedings of International Conference on Software Maintenance, 1994, pp. 120-126.

[23] Jolliffe, I. T., *Principal Component Analysis*, Springer Verlag, 1986.

[24] Komondoor, R. and Horwitz, S., "Finding duplicated code using program dependences", in Proceedings of European Symposium on Programming, Genoa, Italy, April 2-6 2001.

[25] Kontogiannis, K., "Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics", in Proceedings of Working Conference on Reverse Engineering, Amsterdam, The Netherlands, October 6-8 1997, pp. 44-55.

[26] Kontogiannis, K., Galler, M., and DeMori, R., "Detecting code similarity using patterns", in Proceedings of Third Workshop on AI and Software Engineering : Breaking the Toy Mold, August 1995, pp. 68-73.

[27] Lague, B., Proulx, D., Merlo, E., Mayrand, J., and Hudepohl, J., "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", in Proceedings of International Conference on Software Maintenance, Bari, Italy, October 1-3 1997, pp. 314-321.

[28] Landauer, T. K. and Dumais, S. T., "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge", *Psychological Review*, vol. 104, no. 2, 1997, pp. 211-240.

[29] Landauer, T. K., Laham, D., Rehder, B., and Shreiner, M. E., "How Well Can Passage meaning Be Derived without Using Word Order? A Comparison of Latent Semantic Analysis and Humans", in Proc of Proceedings of the 19th Annual Conference of the Cognitive Science Society, 1997, pp. 412-417.

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

[31] Maarek, Y. S. and Smadja, F. A., "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proc. of SIGIR89, Cambridge, MA, June 1989, pp. 198-206.

[32] Maletic, J. I. and Marcus, A., "Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding", in Proceedings of 12th IEEE International Conference on Tools with Artificial Intelligence, Vancouver, British Columbia, Nov. 13-15 2000, pp. 46-53.

[33] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings of 23rd International Conference on Software Engineering, Toronto, Ontario, May 12-19 2001, pp. 103-112.

[34] Maletic, J. I. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis", in Proc. of 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach FL, Oct. 1999, pp. 251-254.

[35] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R., "Using Automatic Clustering to Produce High-Level Organization of Source Code", in Proc of 6th International Workshop on Program Comprehension, Italy, June 1998.

[36] Mayrand, J., Leblanc, C., and Merlo, E., "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", in Proceedings of International Conference on Software Maintenance, Monterey, CA, November 4-8 1996, pp. 244-254.

[37] Michail, A. and Notkin, D., "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships", in Proceedings of International Conference on Software Engineering, 1999.

[38] Mosaic, "Mosaic Source Code v2.7b5", NCSA site, Date Accessed: 4/12/00, ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/.

[39] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S., "A Reverse Engineering Approach to Subsystem Structure Identification", *Software Maintenance: Research and Practice*, vol. 5, no. 4, 1993, pp. 181-204.

[40] Ning, J. Q., Engberts, A., and Kozaczynski, W., "Recovering Reusable Components from Legacy Systems", in Proc. of Working Conference on Reverse Engineering, 1993.

[41] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 1996.

[42] Rieger, M. and Ducasse, S., "Visual Detection of Duplicated Code", *Object-Oriented Technology (ECOOP'98 Workshop Reader)* July 1998, pp. 75-76.

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

[44] Schwanke, R. W., "An intelligent tool for re-engineering software modularity", in Proceedings of 13th International Conference on Software Engineering, 1991, pp. 83-92.

[45] Strang, G., *Linear Algebra and its Applications*, 2nd ed., Academic Press, 1980.

[46] Wiggerts, T., "Using clustering algorithms in legacy systems remodularization", in Proceedings of Working Conference on Reverse Engineering, 1997, pp. 33-43.