DETECTING DOMAIN FROM SOURCE CODE USING SEMANTIC CLUSTERING

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

Thapar University, Patiala

By:
SANJAY MADAN
(80732016)

Under the supervision of:
Ms. Shalini Batra
Lecturer (SS)

MAY 2009

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Detecting Domain from Source Code Using Semantic Clustering"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Ms. Shalini Batra and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(SANJAY MADAN)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Ms. SHALINI BATRA)
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. SEEMA BAWA)
Professor & Head
Computer Science & Engineering Department
Thapar University,
Patiala

(Dr. R.K.SHARMA) 7/7/09
Dean (Academic Affairs)
Thapar University,
Patiala.

ABSTRACT

To understand the software source code lots of approaches have been developed and many of them concern to the program structural information but this results in the loss of domain semantic crucial information contained in the text or symbols of source code. To understand software as a whole, we need to enrich these approaches with conceptual insights gained from the domain semantics. This thesis proposes the mapping of domain to the code using the information retrieval techniques to use linguistic information, such as identifier names and comments in source code. Here we introduce an algorithm based on the concept of Semantic Clustering to group source artifacts based on how the synonymy and polysemy is related. The algorithm uses the concept of Latent Semantic Indexing (LSI). The biggest advantage of the approach used is that it works at the source code textual level thus making it language independent. It correlates the semantics with structural information applies at different levels of abstraction (e.g. packages, classes, methods).

After detecting the clusters, based on semantic similarity automatic labeling of the program code is done and is visually explored. Since 3-Dimension visualization makes the concept detection much easier, we have concentrated on visualization of semantic clusters detected in the source code.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Technology is changing at the speed of light every now and then. Software Re-engineering is a course of action in which an analysis of existing software system is done and due modifications are carried out to constitute a new form. The concept of reverse-engineering aims at understanding the functionality and implementation of existing software, and re-implementing the newer version, to improve the system's functionality and performance. To get knowledge about a software system is one of the main activities in software reengineering, majority of software maintenance is spent on comprehension [1]. This is because a lot of knowledge about the software system and its associated business domain is not captured in an explicit form. Most approaches that have been developed focus on program structure [2] or on external documentation [3, 4]. Other approaches using methods such as dynamic data or history information have also proven valuable. However, the identifier names and the source code comments are the main fundamental source of information.

Developers use meaningful identifier names as mnemonics to remember the purpose of code items, to document the code, and to communicate the meaning of their code with other team members or even other teams. These names are a human readable description conveying the meaning and purpose of code items.

The Source code comprises of two types of communication: human-machine communication through program instructions and human to human communications through names of identifiers and comments [5]. The executables are for machine where as code is written for humans not for machines. Let us consider a small code example, which tell whether a time value is in the morning:

```
/** Return true if the given 24-hour time is in the morning and false otherwise. */
public boolean isMorning(int hours,int minutes,int seconds) {
  if (!isDate(hours, minutes, seconds)) throw Exception(”Invalid input: not a time value.”)
                return hours < 12 && minutes < 60 && seconds < 60; }
```

When we strip away all identifiers and comments, from the machine point of view the functionality remains the same, but for a human reader the meaning is obfuscated and almost impossible to figure out. In our example, retaining formal information yields:

```
public type1 method1(type2 a, type2 b, type2 c) {
        if (!method 2(a, b ,c)) throw Exception(literal 1).
                return (a < A) && (b < B) && (c < C); }
```

In this informal information, the vocabulary is presented in random order and the domain of the code is still recognizable. In our example, retaining only the naming yields:

```
is int hours minutes int < minutes input hours is
seconds && boolean morning false 24 time minutes not
60 invalid && value seconds time < seconds hour
given hours 60 12 < morning date int is otherwise
```

Information retrieval provides means to analyze, classify and characterize text documents based on their content. They are used to derive topics from the vocabulary usage at the source code level. Apart from external documentation, the location and use of source-code identifiers is the most frequently consulted source of information in software maintenance. The given representation of documents as bag-of-terms is a well-established technique in information retrieval (IR), and is used to model documents in a text corpus. In the software analysis different approaches that apply IR on external documentation [6, 7], but only few work has been focused on treating the source code itself as data source.
Latent Semantic Indexing (LSI) is an information retrieval technique that locates linguistic topics in a set of documents [8, 9]. LSI is applied to compute the linguistic similarity between source artifacts (e.g. packages, classes or methods) and cluster them according to their similarity. Latent semantic indexing (LSI) is used to exploit linguistic information from the source code contained in the names of identifiers and the content of comments, this result in a search index on software artifacts. With this index, the software system is grouped into clusters. This work is based upon the Semantic Clustering approach, a novel technique to characterize the semantics of a software system. It is a non-interactive and unsupervised technique. A semantic cluster is a group of artifacts that use same vocabulary, so the each cluster reveals a different concept found in the system.

Finally, automatic algorithms labels each cluster with its most related terms and provide the human readable description of the main concepts in a software system.

Three steps of the topic identification from source code include: pre-processing, applying LSI, and clustering. Furthermore we retrieve the most relevant terms for each cluster, thus in short the approach is:

(1) **Pre-processing the software system.** Break the system into documents and build a term-document-matrix that contains the vocabulary usage of the system.

(2) **Applying Latent Semantic Indexing.** Use LSI to compute the similarities between source code documents and illustrate the result in a correlation matrix [10].

(3) **Identifying topics.** Cluster the documents based on their similarity, we rearrange the correlation matrix and each cluster is a linguistic topic.

(4) **Describing the topics with labels in 3-Dimension.** Use LSI again to retrieve the top n most relevant terms for each cluster.

1.1 Structure of the Thesis

The thesis is divided into six chapters:

Chapter 1 – The introduction to the work is provided.

Chapter 2 – Literature review section gives a brief survey of research going on in the area of reverse engineering, semantic clustering and enhancements required thereof.

Chapter 3 - After going through literature review problem statement has been identified and defined in this chapter.

Chapter 4 - The general concepts used in the thesis work are introduced. This chapter serves as theoretical background for future work.

Chapter 5 - An evaluation model of work done is described. The illustrations captured during the work done are presented.

Chapter 6 – Focuses on the conclusion of the work presented in this thesis and ideas for further research are given.

At the end of thesis, the appendix and references are provided.

Chapter 2

LITERATURE REVIEW

Analysis of the existing software system and modifying the process as per the requirement is known as software reengineering. It is the process of nurturing the software system at structural and logical level for the future up gradation and for the system understanding. Domain detection from existing source is part of it. In this section, work done in the area of domain detection from source code is reviewed and focus has been made on the visualization of domain concept using the 3-Dimensional software system visualization approach. The fundamental concept for domain detection is to retrieve the linguistic information from the source code and cluster the semantically similar terms. LSI [20] (Latent Semantic Indexing) is an information retrieval (IR) techniques based upon the vector space model (VSM) [27] approach. This approach models documents as bag-of-words and arranges them in a Term-Document Matrix.

The IR free-text approach consists of drawing information only from the structure of some documents which provide information about the software components. No semantic knowledge is used and no interpretation of the document is given. The reuse tool attempts to characterize the document rather than understand it. There are approaches in the field of software analysis that apply IR on external documentation [4, 6], but only few work has been focused on treating the source code itself as data source. Caprile and Tonella analyzed the lexical, syntactical and semantical structure of function identifiers in [17]. The main difficulty found is that source code has a quite scarce and ambiguous vocabulary compared to natural language documents. Marcus et al proposed Latent Semantic Indexing (LSI), an information retrieval technique that takes synonymy and polysemy into account, as a means to locate concepts [9, 11]. Recently Kawaguchi et al used LSI and clustering to categorize software projects at the level of entire projects [12].

This work is based on both of these approaches, and goes beyond them. It decomposes the system into its main semantic concepts, and characterizes each concept with automatically retrieved labels. It is applicable at any level of abstraction, such as: packages, classes and methods, or even program slices like execution traces [24]. Additionally, it introduces two visualizations: one focused on the correlation among semantics and one focused the relation between semantics and structure.

The clustering from the most stable algorithm bears little similarity to the implemented system structure, while the clustering from the least stable algorithm has the best cluster distribution. From the results of Semantic clustering, we claim that current automatic semantic clustering algorithm need significant improvement to provide continual support for large software projects [25]. Determining the input of the clustering process and specifying criteria for good clustering will depend heavily on the software environment used. As a software system will continue to evolve after it has been remodularized, incrementally updating a modularization requires special attention. Not much research has been done in this area however.

Distribution Map [26] as a generic technique to visualize and categorize existing entities into new groups or associates them with mutually exclusive properties. The technique is based on the notion of focus, which shows whether a property is well encapsulated or cross-cutting, and the notion of spread, which shows whether the property is present in several parts of the system. This present a basic visualization and complement it with measurements that quantify focus and spread. A major limitation of the visualization of the system as distribution map is that the approach is based upon boxology as the result the clustering labels are not fully visualized and some of the crucial information is missing.

Chapter 3

PROBLEM STATEMENT

3.1 Problem Definition

During software development and evolution most activities involve changes to the existing source code. To carry out such tasks, software engineers spend a lot of time identifying the places where changes are to be made. Source code searching and browsing are two of the most common activities undertaken by developers, especially during maintenance of existing large software. These activities directly support tasks such as concept location in source code, impact analysis, change propagation, and comprehension in general.

Concept location is traditionally an intuitive and informal process, based on the past experience with the system. The process of concept location is to find the code that implements these concepts. In addition to problem domain concepts, change requests can also be formulated in terms of the solution domain. A description of a problem or solution domain concept expressed in natural language is the input to the concept location process. The output is a set of software components that implement or address the concept. The traceability links between external documentation and the source code provide all that is needed for concept location.

During the literature review it was analyzed that concept location is implement in number of ways but it is not clearly evident and the visualization of the concepts will lead to automated and dynamic computations over the system and semantic clustering will conceptualize the data as the similar words are in the same cluster of documents.
To overcome interoperability and association of the classes in semantically cluster environment, different visualization of the system is required which can clearly visualize the whole text corpus with complete system association. The work done in this topic visualizes the system in boxology based technique, which is flat cluster visualization technique.

3.2 Methodology

The step-by-step methodology to be followed for domain detection from Source code using semantic Clustering as follows:

- A thorough analysis of various Information retrieval techniques from Source Code.
- Study of LSI technique for information retrieval and implement the algorithm using Semantic Clustering.
- Automate the clusters to identify the topic and implement automatic labelling algorithm for clusters.
- Visualize the system in 3-Dimension to analyze the advantages and disadvantages on different parameters.
- Comparison of the results achieved after the execution of program with the earlier outputs.

Chapter 4

Semantic Clustering

The goal of any information retrieval (IR) system is to identify documents relevant to a user's query. In order to do this, an IR system must assume some specific measure of relevance between a document and a query, i.e., an operational definition of a relevant document with respect to a query. A fundamental problem in IR research is to formalize the concept of relevance; a different formalization of relevance generally leads to a different retrieval model.

Over the decades, many different retrieval models have been proposed, studied, and tested. Their mathematical basis spans a large spectrum, including algebra, logic, probability and statistics. The existing models can be roughly grouped into three major categories, depending on how they define/measure relevance. In the first category, relevance is assumed to be correlated with the similarity between a query and a document. In the second category, a binary random variable is used to model relevance and probabilistic models are used to estimate the value of this relevance variable. In the third category, the relevance uncertainty is modelled by the uncertainty of inferring queries from documents or vice versa. Here we consider only the first category in detail.

4.1 Latent Semantic Indexing

Latent Semantic Indexing (LSI) [20] is a machine-learning model that induces representations of the meaning of words by analyzing the relation between words and passages in large bodies of text. LSI has been used in applied settings with a high degree of success in areas like automatic essay grading and automatic tutoring to improve summarization skills in children. As a model, LSI's most impressive achievements have been in human language acquisition simulations and in modeling of high-level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity.

LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with vector space model (VSM) [27] approaches. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-

occurrence matrix decomposed using Singular Value Decomposition. It is an information retrieval technique that locates linguistic topics in a set of documents [8, 9]. LSI is applied to compute the linguistic similarity between source artifacts (e.g. packages, classes or methods) and cluster them according to their similarity. This clustering partitions the system into linguistic topics that represent groups of documents using similar vocabulary.

Latent Semantic Indexing (LSI), is a technique common in information retrieval to index, analyzes and classifies text documents. It analyzes how terms are spread over the documents of a text corpus and creates a search space with document vectors: similar documents are located near each other in this space and unrelated documents far apart of each other. It is used to analyze the linguistic information of a software system as the source code is basically composed of text documents. It has been shown in [15, 20] that LSI addresses the synonyms very well. With simple corpus training, LSI managed to answer correctly 64% of the synonyms questions in the Test of English as a Foreign Language, better than the average student.

There is a wide range of applications of LSI, such as automatic assignment of reviewers to submitted conference papers [10], cross-language search engines, spell checkers and many more. In the field of software engineering LSI has been successfully applied to categorized source files [11] and open-source projects [12], detect high-level conceptual clones [13], recover links between external documentation and source code [14,15]. Furthermore LSI has proved useful in psychology to simulate language understanding of the human brain, including processes such as the language acquisition of children.

All techniques for concept location reduce the search space that the user needs to review. However, the user is still necessary in order to locate an actual concept in the code.

4.1.1 Preparing the Corpus and LSI Space

Domain knowledge and concepts are embedded in the source code through identifier names and internal comments. We target these elements from the source code to be analyzed by LSI; therefore a simple pre-processing of the source code is needed. Three actions are taken here:

1. Extraction of identifiers and comments
2. Identifier separations
3. Establishing document granularity.

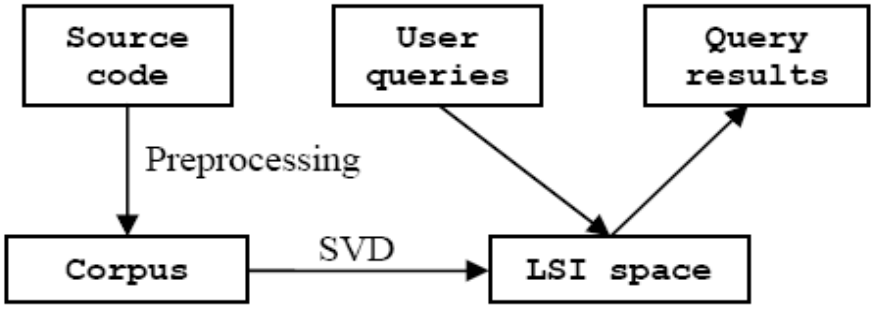

Figure 4.1: The concept location process using LSI [9].

Extraction of identifiers and comments requires very limited parsing and similarity among the programming languages allows developing a tool that deals with several languages. The next step involves identifier separation. While not paramount with respect to the results, it is a simple step that enriches the corpus and improves the results. There are two commonly used coding styles for identifiers: one is the combination of words using underscore "_" as separators (e.g., concept_location) and the other is the combination of words using letter capitalization for saparation (e.g., ConceptLocation, CONCEPTLocation). All identifiers that follow these rules are separated into constituent words. The original form of the identifier is also maintained and the separated words are added into the corpus immediately following the identifier and will be later processed by LSI.

The final step of the pre-processing is partition of the code into documents. For systems written in procedural languages we choose each function to be a separate document and all declarations blocks outside functions in each file to be treated as one document each [9].

4.1.2 Term-Document Matrix

Like other information retrieval (IR) techniques, Latent Semantic Indexing (LSI) is based on the vector space model (VSM) approach. This approach models documents as bag-of-words and arranges them in a Term-Document Matrix A, such that $a_{i,j}$ equals the number of times term t_i occurs in document d_j.

As discussed earlier, LSI can overcome problems with synonymy and polysemy that used to occur in prior vectorial approaches, and thus improves the basic vector space model by replacing the original term-document matrix with an approximation. This is done using singular value decomposition (SVD), a principal components analysis (PCA) technique

originally used in signal processing to reduce noise while preserving the original signal. Assuming that the original term-document matrix is noisy (the synonymy and polysemy), the approximation is interpreted as a noise reduced – and thus better – model of the text corpus.

For example, a typical search engine covers a text corpus with millions of web pages, containing some ten thousands of terms, which is reduced to a vector space with 200-500 dimensions only. In Software Analysis, the number of documents is much smaller and we reduce the text corpus to 20-50 dimensions.

The entire corpus can be represented as a term-document matrix D. D consists of document vectors d_i. The document vectors are the columns of this matrix. Document vector d_i contains m generalized terms.

$$D = \begin{bmatrix} | & | & \dots & | \\ \vec{d}_1 & \vec{d}_2 & \dots & \vec{d}_n \\ | & | & \dots & | \end{bmatrix}, \quad D \in \mathbb{R}^{m \times n}.$$

Since the terms might not be of the same type (i.e. one term may be a term count, one may be some relative quantity) it no longer makes sense to normalize across terms. So, it is assumed that the columns of D are not normalized document vectors, because they are considered as generalized terms. Earlier the document vectors were considered individually. The question of how the documents in the corpus relate to each other is not considered, except implicitly with respect to the query and in consideration of the IDF (Inverse Document Frequency). The idea is to get a better representation of the documents that corresponds to the structure of the corpus, since a better representation may help us do better retrieval.

Inverse document frequency (IDF) considers the whole document corpus. IDF looks at the distribution of the terms in the whole corpus and shrinks term frequencies for terms that occur a lot in the corpus. IDF thus uses the overall corpus characteristics to alter our document representation.

Figure 4.1 schematically represents the LSI process. The document collection is modelled as a vector space. Each document is represented by the vector of its term occurrences, where terms are words appearing in the document. The term-document-matrix A is a sparse matrix and represents the document vectors on the rows. This matrix is of size n × m, where m is the number of documents and n the total number of terms over all documents. Each entry $a_{i,j}$ is the frequency of term t_i in document d_j . A geometric interpretation of the term-document-matrix is a set of document vectors occupying a

vector space spanned by the terms. The similarity between documents is typically defined as the cosine or inner product between the corresponding vectors. Two documents are considered similar if their corresponding vectors point in the same direction.

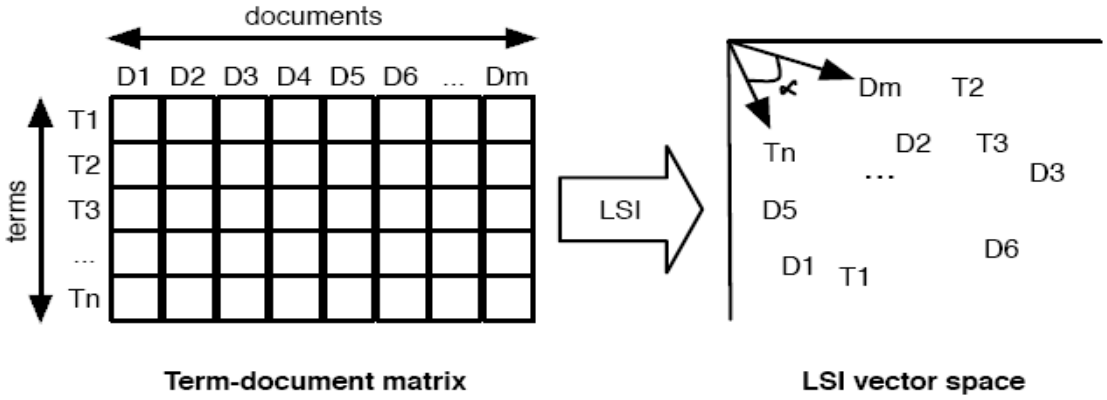

Figure 4.2, LSI takes as input a set of documents and the terms occurrences, and returns as output a vector space containing all the terms and all the documents. The similarity between two items (i.e., terms or documents) is given by the angle between their corresponding vectors [5].

4.1.3 Data Normalization

All words are not equally significant as some words are more likely to discriminate documents than others. Words with a high frequency are considered to be too common and those with low frequency too rare, and therefore both of them are not contributing significantly to the content of a document. Words with medium frequency have the highest ability to discriminate content [28]. Thus all words above an upper and all words below a lower threshold are excluded from the term list.

The removal of high frequency words is usually done excluding a list of common words, called stopwords. As the distribution of word frequencies follows the power law [29], this removal reduces the size of a text corpus by about 30 to 50 percent. In case of an English text corpus, the SMART stopword list is well-established: it contains about 500 common non-discriminative words like the, of, to, a, is [30]. The removal of low frequency words is usually done by excluding all words that occur in only one document.

Furthermore the same term may appear in different grammatical inflections. Most words are composed of a stem, which bears the meaning of the word, and a suffix that bears grammatical information. If two words have the same stem then they refer to the same concept and should be indexed as one term. The process of removing the grammatics is called stemming, and a standard approach is to have a list of suffixes and to remove the longest possible one. For example, train, trained and training are all reduced the common

stem train. In case of an English text corpus the Porter Stemming Algorithm is well-established [34].

Research in information retrieval (IR) has shown that normalization leads to more effective retrieval than if the raw term frequencies were used [35].

4.1.4 Term Frequency and Weighting

The goal of a weighting function is to balance out very rare and very common terms. Typically a combination of two weighting schemes is applied, a local weighting and a global weighting. The first puts a term occurrence in relation to its document, and the latter in relation to the whole text corpus. IR systems assign weights to terms by considering:

1. Local information from individual documents
2. Global information from collection of documents

In addition, systems that assigns weights to link graph information to properly account for the degree of connectivity between documents. This weighting scheme is given by Equation:

$$\text{Term Weight} = \quad w_i = tf_i * log\left(\frac{D}{df_i}\right) \qquad \text{.....(4.1.1)}$$

Where

- tf_i = term frequency (term counts) or number of times a term i occurs in a document.
- df_i = document frequency or number of documents containing term i
- D = number of documents in the database

4.1.4.1 Local Weights

The above equation, 4.1 shows that w_i increases with tf_i. This makes the model vulnerable to term repetition. Given a query q,

1. For documents of equal lengths, those with more instances of q are favoured during retrieval.
2. For documents of different lengths, long documents are favoured during retrieval since these tend to contain more instances of q.

4.1.4.1 Global Weights

In above equation, the $\log(D/df_i)$ term is known as the inverse document frequency (IDF_i) – a measure of the shear volume of information associated to a term i within a set of documents. Inspecting the df_i/D ratio, this is the probability of retrieving from D a document containing term i. In equation, we simply invert this probability and take its log. The result is then premultiplied by tf_i. Several modifications to the equation have been proposed and more general form of this equation is given below:

$$x_{i,j} = \text{local}(t_i, d_j) \text{ X } \text{global}(t_i) \quad \text{.....(4.1.2)}$$

Equation 4.1 shows that w_i decreases as df_i increases. For example, if in a 1000-document database only 10 documents contain a particular term, the IDF for this term is log(1000/10) = 2. However, if only one document contains that term, then IDF is log(1000/1) = 3. Thus, terms which appear in too many documents (e.g. stopwords, very frequent terms) receive a low weight, while uncommon terms which appear in few documents receive a high weight. This makes sense since too common terms are not very useful for distinguishing a relevant document from a non-relevant one. Terms with acceptable weights are those that are not too common or too rare; i.e. their term vectors are not too far or too close to the query vector.

When applied on textual data LSI achieves best results with the entropy weighting. Nevertheless we present here the tf-idf -weighting as it is more popular in both information retrieval and software analysis.

Td-ifd stands for “term frequency – inverted document frequency”, and divides the frequency of a term by the number of documents that contain this term, that is locally used terms weight more then globally used terms. After tf-idf normalization the resulting elements of A become:

$$a_{i,j}^{tfidf} = \log(a_{i,j}) \times \log \frac{|D|}{df_i} \quad \text{..... (4.1.3)}$$

Where df_i is the number of documents that contain term t_i and |D| is the total number of documents. Furthermore, the length of the documents vectors is usually normalized too.

4.1.5 Singular Value Decomposition

LSI starts with an input as term-document-matrix, weighted by a weighting function to balance out very rare and very common terms. SVD is used to break down the vector space model into less dimensions. This algorithm preserves as much information as possible about the relative distances between the document vectors, while collapsing them into a much smaller set of dimensions. This downsizing is achieved using singular value decomposition (SVD), a kind of principal component analysis originally used signal processing to reduce noise while preserving the actual signal. Assuming that the original term-document matrix is noisy (synonymy and polysemy), the approximation A_k is interpreted as a noise reduced – and thus better model of the text corpus.

SVD decomposes matrix A into its singular values and its singular vectors, and yields – when truncated at the k largest singular values – an approximation A` of A with rank k. Furthermore, not only the low-rank term-document matrix A` can be computed but also a term-term matrix and a document-document matrix. Thus, LSI allows us to compute term-document, term-term and document-document similarities.

$$A = U \times S \times V^T$$

Singular value decomposition transforms the matrix A into three matrices, using eigenvalue decomposition. This yields three matrices: $U \times S \times V^T$. The two outer matrices contain the singular vectors: U is the term matrix, it contains the left singular vectors and each of its row corresponds to a term. The same goes for V, the document matrix, which contains the right singular vectors and whose rows correspond to documents. The middle matrix S is a diagonal matrix with the singular values, which are a kind of eigenvalue, of A in descending order.

When multiplying all three matrices together to reconstruct the original matrix, the singular values act as weights of the singular vectors. A singular vector with a large corresponding singular vectors has a large impact on the reconstruction, while a small value indicates a singular vector with almost no impact on the result. Thus small values and their vectors may be discarded without affecting the result noticeably. Keeping the k largest singular values only yields a low-rank approximation of A, which is the best rank k approximation of A under the least-square-error criterion:

$$A`_{nxm} = U_{nxk} \times S_{kxk} \times V^T_{kxm}$$

As the rank is the number of linear-independent rows and columns of a matrix, the vector space spanned by A` is of dimension k only and much less complex than the initial space. Applying SVD on natural data (such as signals, images or text documents) yields a distribution of singular values that follows the power law: a few large values, and a long tail with very small values [29]. When used for information retrieval, k is typically about 200-500, while n and m may go into millions. That is why a text corpus with millions of documents can be approximated with such a low-ranked matrix. When used to analyze software on the other hand, k is typically about 20−50 with vocabulary and documents in the range of thousands only, and since A` is the best approximation of A under the least-square-error criterion, the similarity between documents is preserved, while in the same time mapping semantically related terms on one axis of the reduced vector space and thus taking into account synonymy and polysemy. In other words, the initial term-document-matrix A is a table with term occurrences and by breaking it down to much less dimension the latent meaning must appear in A` since there is now much less space to encode the same information. Meaningless occurrence data is transformed into meaningful concept information.

4.1.6 Term and Documents Similarity

To show the SVD factors geometrically, the rows of the matrices are taken as coordinates of points representing the documents and terms vector dimensional space as shown in Figure 4.3 The nearer one points to the other, if they are more similar documents or terms (see Figure 4.3). Similarity is typically defined as the cosine between the corresponding vectors:

$$sim(di, dj) = cos(vi, vj)$$

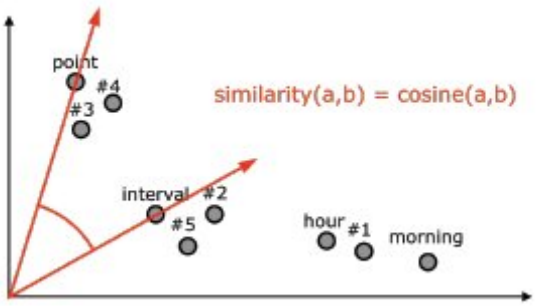

Figure 4.3: On the left: An LSI-Space with terms and documents, similar elements are placed near each other [5].

Computing the similarity between document di and dj is done taking the cosine between the i-th and j-th row of the matrix.

The resulting cosine value, similarity values range from 1 to 0: 1 for similar vectors with the same direction and to 0 for dissimilar, orthogonal vectors. Theoretically cosine values can go all the way to −1, but because there are no negative term occurrences, similarity values never goes below to zero.

4.2 Semantic Clustering

Semantic clustering is a non-interactive and unsupervised technique to analyze the semantics of a software system. Semantic clustering offers a high level view on the domain concepts of a system, abstracting concepts from software artifacts. Firstly, Latent Semantic Indexing (LSI) is used to extract linguistic information from the source code and then clustering is applied to group the related software artifacts into clusters and groups of artifacts having the same vocabulary are identified and these are called clusters. Thus each cluster reveals a different concept of the system. Most of these are domain concepts, some are implementation concepts. The actual ratio depends on the naming convention of the system.

At the end, the inherently unnamed concepts are labeled with terms taken from the vocabulary of the source code. An automatic algorithm labels each cluster with most similar terms, and in this way provide a human readable description of the main concepts in a software system. Additionally, the clustering is visualized as a shaded Correlation Matrix that illustrates:

- The semantic similarity between elements of the system, the darker a dot the more similar its artifacts,
- A partition of the system into clusters with high semantic cohesion, which reveals groups of software artifacts that implement the same domain concept,
- Semantic links between these clusters, which emphasize single software artifacts that interconnect the above domain concepts.

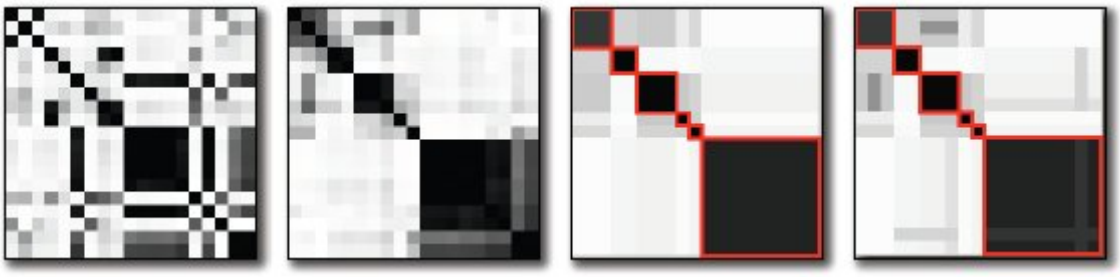

Figure 4.4: From left to right: unordered correlation matrix, then sorted by similarity, then grouped by clusters, and finally including semantic links [5].

4.2.1 Building the Text Corpus

T**ext corpus** is a large and structured set of texts. To build a semantic model, Latent Semantic Indexing (LSI) is used to analyze the distribution of terms over a text corpus. When applying LSI on a software system we will break its source code into documents and use the vocabulary found therein as terms. The system can be split into documents at any level of granularity, such as modules, classes or methods, it is even possible to use entire projects as documents [16].

The vocabulary of source can be extracted both from the content of comments and from the identifier names. Comments are parsed as natural language text and compound identifier names split into their parts. As most modern naming conventions are used camel case, it is straight forward to split identifiers: for example, FooBar becomes foo and bar. In case of legacy code that uses other naming conventions, more advanced algorithms and heuristics are required [17, 18].

Common stopwords are excluded from the vocabulary, as they do not help to discriminate documents, and stemmer algorithm is used to reduce all words to their morphological root. Finally the term-document matrix is weighted with tf-idf (Term frequency, inverted document frequency), to balance out the influence of very rare and very common terms.

4.2.2 Semantic Similarity

Semantic similarity is the likeness of meaning/semantic content within a set of documents or terms. Latent Semantic Indexing (LSI) can be used to extract linguistic information from the source code. The result of this process will be an LSI index L with similarities between software artifacts as well as terms. Based on the index the similarity between these elements can be determined. Software artifacts are more similar if they cover the same concept, terms are more similar if they denote related concepts. Since similarity is defined as cosine between element vectors, its values range between 0 and 1. The similarities between elements are arranged in a square matrix A called the Correlation Matrix.

To visualize the similarity values, it has been mapped to grey values, the dark ones i.e. the one which are dense are more similar. In that way the matrix becomes a raster-graphic with gray dots: each dot $a_{i,j}$ shows the similarity between element d_i and element d_j . The elements are arranged on the diagonal and the dots in the off-diagonal show the relationship between them.

4.2.3 Semantic Clustering

Without proper ordering, the correlation matrix looks like a television tuned to a dead channel. An unordered matrix does not reveal any patterns: arbitrary ordering, such as the names of the elements, is generally as useful as random ordering [19]—therefore, matrix will be clustered such that similar elements are put near each other and dissimilar elements far apart of each other. After applying the clustering algorithm, the similar elements are grouped together and aggregated into concepts. Hence, a concept is characterized as a set of elements that uses the same vocabulary. Documents that are not related to any concept usually end up in singleton clusters in the middle or in the bottom right of the correlation matrix. The correlation matrices are ordered using average linkage clustering algorithm.

The matrix will be reordered first, and then dots will be grouped by clusters and colour them with their average cluster similarity. As with the element similarities in the previous section, the similarities between clusters are arranged in a square matrix A. When visualized, this matrix becomes a raster-graphic with gray rectangles: each rectangle $r_{i,j}$ shows the similarity between cluster R_i and cluster R_j , and has the size ($|R_i|$, $|R_j|$). The clusters are arranged on the diagonal and the rectangles in the off-diagonal show the relationship between them—see the third matrix on Figure 4.4.

4.2.4 Semantic Links

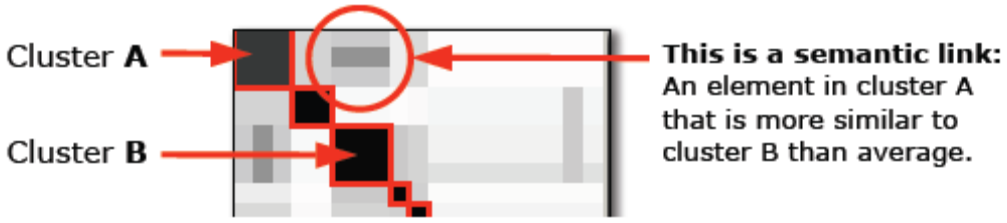

Figure 4.5: A semantic link is a one-to-many relation: A document in cluster A that is more similar to those in cluster B than all its siblings in A [5].

The trade off of the clustering is, as with any abstraction, that some valuable detail information is lost. Semantic linking has been used to pick out relations at the level of elements, and plot them on top of a clustered correlation matrix.

The most valuable patterns are one-to-many relationships between an element and an entire cluster. If the similarity between an element d_n and a cluster differs significantly from the average cluster similarity, the clustered matrix is the output: as a bright line if d_n

is less similar then average, and as a dark line if d_n is more similar then average. Figure 4.5 gives an example of a semantic link. But firstly, the elements are grouped by top-level modules and then clustered within these modules.

4.2.5 Correlation Matrix

A correlation matrix is gray-scale raster-graphic: each dot $a_{i,j}$ shows the similarity between element d_i and element d_j—the darker, the more similar. The elements are arranged on the diagonal while the dots in the off-diagonal show the relationship between them. An unordered matrix does not reveal any patterns, therefore we cluster the elements and sort the matrix: all dots in a cluster are grouped together and are colour with their average similarity, this is semantic cohesion [15]. This offers a high-level view on that system, abstracting from elements to concepts. A sample Correlation Matrix is shown in Figure 4.6. There are three annotated clusters: A, B and C.

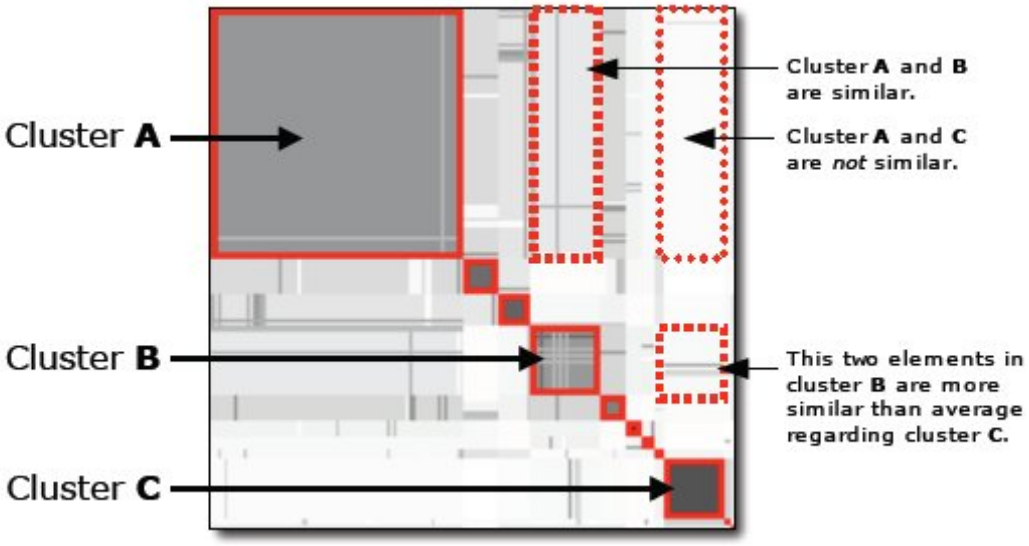

Figure 4.6: A sample Correlation Matrix : Cluster A and B are similar, cluster C is very cohesive, two elements in B are related to C [5].

Semantic Similarity: The colour in the off-diagonal shows the similarity between clusters. Cluster A has much in common with Cluster B as there is a gray area between them, but nothing at all in common with Cluster C since the area between the two is plain white.

Semantic Cohesion: The colour inside a cluster shows the similarity among its elements, that is the "semantic cohesion" of its concept. Cluster C is nearly black and thus very cohesive, cluster A and B are lighter and but still of acceptable cohesion.

Semantic Links. If the similarity between an element d_n and a cluster differs significantly from the average we mark it either with a bright or a dark line. A bright line if d_n is less similar then average, and a dark line if d_n is more similar then average. In Figure 5, there are two elements in cluster B that are more similar than average regarding cluster C.
The Correlation Matrix illustrates the semantics of a software system, which is how the semantic concepts are related to each other.

4.2.6 Labelling the Clusters

Just visualizing clusters is not enough, labelling is required to describe the cluster. Often just enumerating the names of the software artifacts in a cluster gives a sufficient interpretation. If the names are badly chosen or unnamed software artifacts are analyzed, it needs an automatic way to identify labels. Figure 4.7 shows the labels in the concept of LAN example.
The labelling works as follows: there is an LSI-index at hand, use this index as a search engine. Reverse the usual search process where a search query of terms is used to find documents, and instead, use the documents in a cluster as search query to find the most similar term. To obtain the most relevant labels comparison will be performed between the similar terms of the current cluster and similar terms of all other clusters.

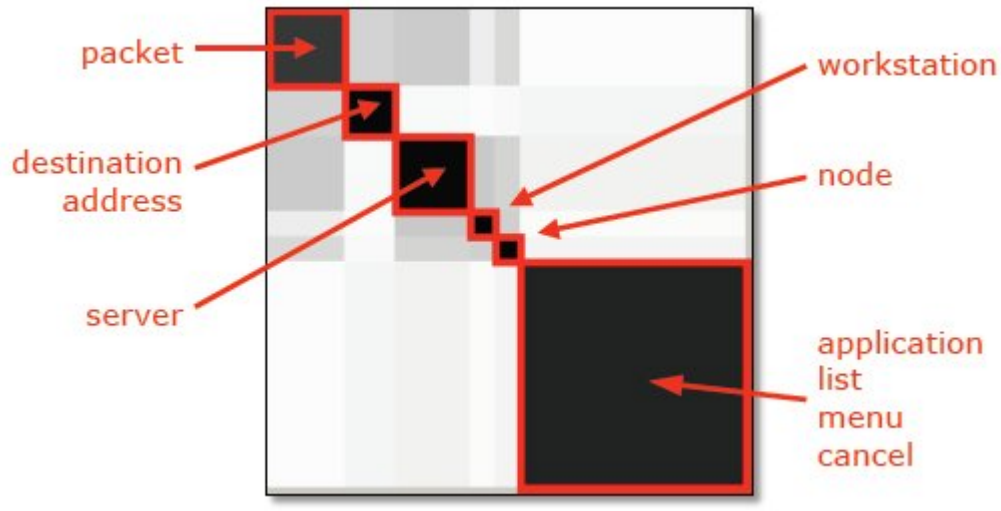

Figure 4.7: Automatically retrieved abels describe the concepts. The labels were retrieved using the documents in a concept cluster as query to search the LSI space for related terms [5].

Term t_0 is relevant to cluster A_0, if it has a high similarity to the current cluster A_0 but not to the remaining clusters A ε $\mathcal{A}$. Given the similarity between a term t and a cluster A as sim(t, A), defines the relevance of term t_0 according to cluster A_0 as follows:

$$rel(t_0, A_0) = sim(t_0, A_0) - \frac{1}{|A|} \sum_{A_n \in A} sim(t_0, A_n) \quad \text{..... (4.2.1)}$$

This raises better results than just retrieving the top most similar terms [23]. It emphasizes terms that are specific to the current cluster over common terms.

4.3 Distribution of Concept in the System

The semantic clusters helps to grasp the concepts implemented in the source code. However, the clustering does not take the structure of the system into account. Distribution Map [19] is the technique to analyze the distribution of the concept over the system. The semantic partition of a system, as obtained by Semantic Clustering, does generally not correspond one-on-one to its structural modularization. In any system finds both, concepts that correspond to the structure as well as concepts that cross-cut it.

4.3.1 Distribution Map

The correlation between the semantic clustering and the structural modularization of a system is represented by Distribution Map, see in Figure 4.8. It is composed of large rectangles containing small squares in different colours. For each structural module there is a large rectangle and within those for each software artifact a small square. The colour of the squares refers to the semantic concepts implements by these artifacts.

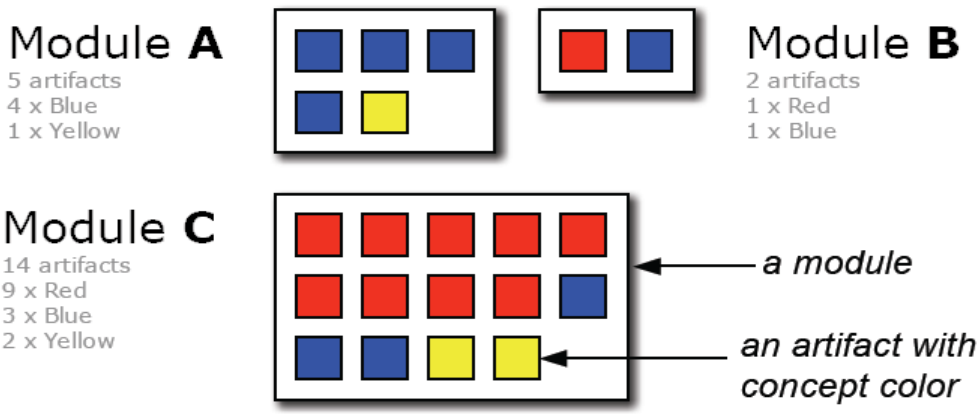

Figure 4.8: The Distribution Map shows how semantic concepts are distributed over the structural modularization of system. Boxes denote modules and software artifacts, colours refer to concepts [5].

The choice of colours is crucial to the readability of the Distribution Map. Scattered squares are easier to spot if they are shown in distinct colours than if they are shown in colours similar to the well-encapsulated colours. Therefore light colours are a good

choice for cross-cutting concerns and dark colours a good choice for well encapsulated concepts. A human reader will draw conclusions concerning the similarity between concepts based on the similarity of the colours that denote these concepts. For example, green and dark green suggests that two concepts are related, while green and red suggest that the same two concepts are unrelated. Therefore the similarity between the semantic concepts is taken into account when choosing the colours, such that more similar concepts use more similar colours and vice versa.

4.3.2 Distribution Patterns

The semantic partition of a system, as obtained by Semantic Clustering, does generally not correspond one-on-one to its structural modularization. In most systems find both, concepts that correspond to the structure as well as concepts that cross-cut it. Given below is a vocabulary to describe the most common distribution patterns (Figure 4.9):

- **Well-encapsulated Concept** – if a concept corresponds to the structure, we call this a well-encapsulated concept. Such a concept is spread over one or multiple modules, and includes almost all artifacts within those modules. If a well-encapsulated concept covers only one module we might prefer to speak of a solitary concept. In Figure 4.9 the red concept illustrates this pattern, but not the orange one.

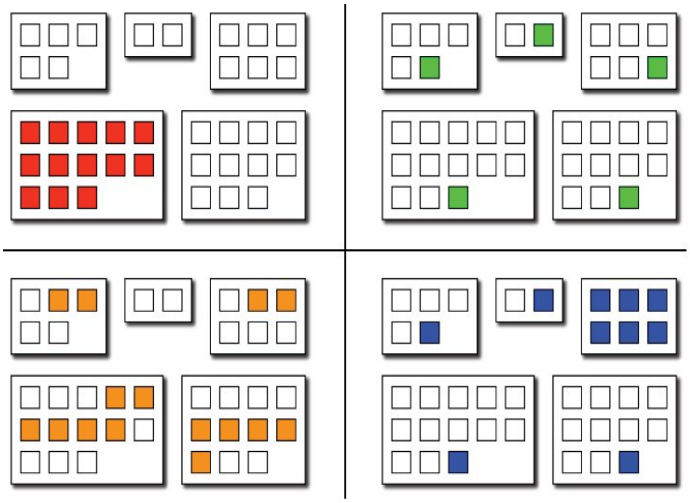

Figure 4.9: From top left to bottom right: a well-encapsulated concept that corresponds to the modularization, a cross-cutting concept, a design smell scattered across the system without much purpose, and finally an octopus concept which touches all modules [5].

- **Cross-Cutting Concepts** – If a concept cross-cuts the structure, that concept is called cross-cutting concept. Such a concept spreads across multiple modules, but includes only one or very few artifacts within each module. In Figure 4.9, the green example illustrates this pattern. Whether a cross-cutting concept has to be considered a design smell or not depends on the particular circumstances. Consider for example the popular three-tier architecture: It takes the concepts accessing, processing and presenting data and puts each on a separate layer; while application specific concepts – such as for example accounts, transactions or customers – are deliberately designated to cross-cut the layers. That is, it picks out some concepts, emphasizes them and deliberately designates the others as cross-cutting concerns.
- **Octopus Concept** – if a concept dominates one module, as a solitary does, but also spreads across other modules, as a cross-cutter does, we call this an octopus concept. In Figure 4.9, the blue example illustrates that pattern. Imagine for example a framework or a library: there is a core module with the implementation and scattered across other modules there are artifacts that plug into the core, and hence using the same vocabulary as the core.
- **Black Sheep** – if there is a concept that consists only of few separate artifacts, we call this a black sheep. Each black sheep deserves closer inspection, as these artifact are sometimes a severe design smell. Yet as often, a black sheep is just an unrelated helper classes and thus not similar enough to any other concept of the system.

The entire process of domain detection from source code described above in detail cam be summarized in the form of diagram in Figure 4.10 and implementation details of this are discussed in the next chapter of the thesis.

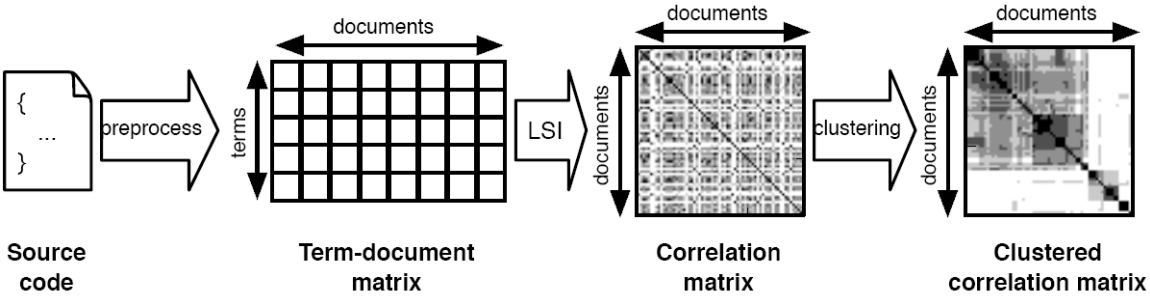

Figure 4.10: Steps for the Semantic Clustering of Source Code [5].

4.4 Tools Used for Semantic Cluster Visualization

Various visualization tools are provided by research community for analyzing, designing and implementing the semantic clustering for the concept location from the source code. Major analyzing tools considered so far include Hapax, Code City and Softwarenaut. These tools visualize the software system and also provide the scripting facility. The brief description of tools used is given below:

- **Hapax:** Hapax is an intuitive and analyzing tool built on top of the Moose reengineering environment. It is a tool which implements and visualizes the clusters and provides the correlation matrix of semantically cluster concepts.
- **Code City:** Code city tool visualizes the system in 3-dimension, which is also built on the top of Moose reengineering environment. It is the tool which analyze, script and visualize the semantically cluster matrix in 3-dimensional format to ease the concept location from the system.

Chapter 5

IMPLEMENTATION AND RESULTS

In the course of detecting domain from source code the reverse engineering technique is applied on the informal information. The reverse engineering has several interactive complementary visualization perspectives of the module structure of a system. The system is visualized in different environment to study the different perspective. Moreover, the visualizations are supported by complementary information in the form of software metrics and other semantic information about the currently inspected part of the system. The implementation for the domain detection is done on Hapax, a Moorse reengineering environment and the algorithm implementation is done on NetBeans, an open source Java based editor. Figure 5.1 shows the environment of NetBeans and program running process. The steps of implementation are given below:

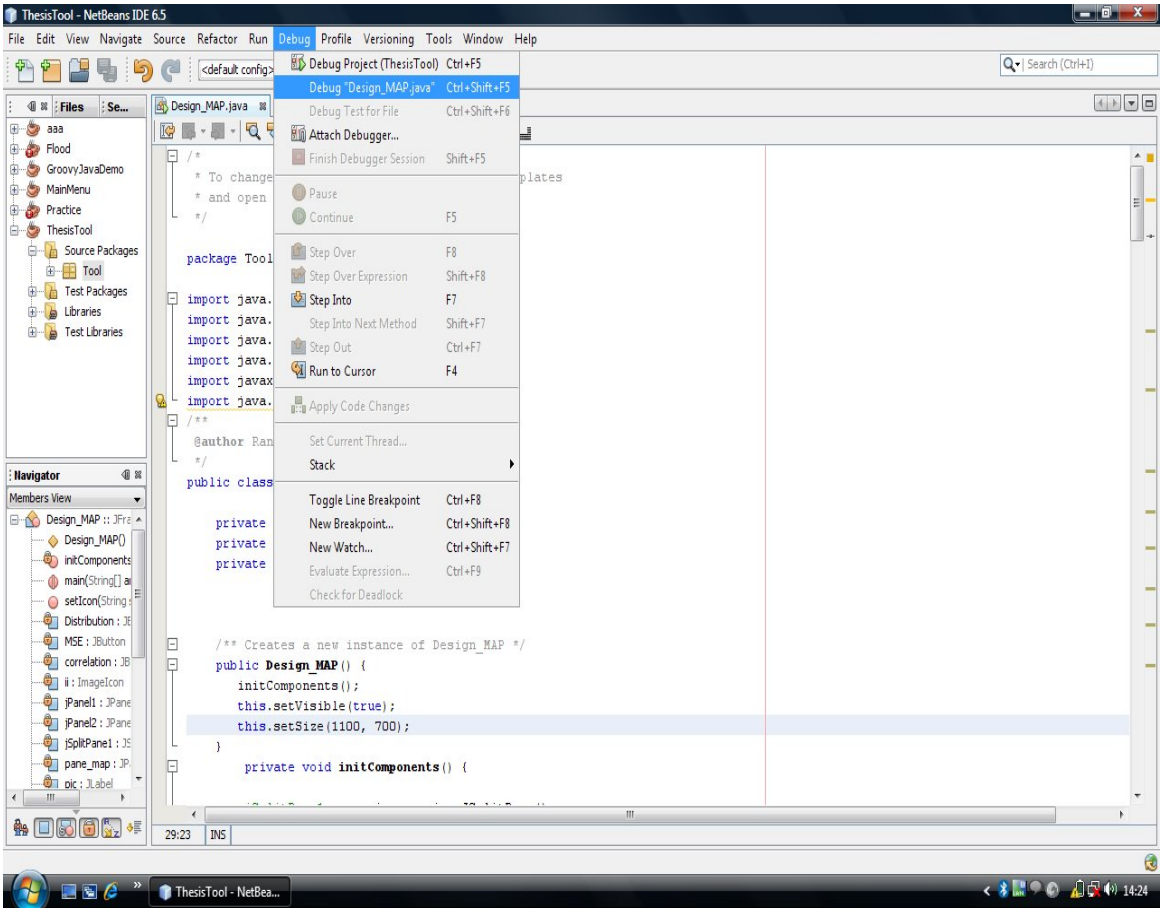

Figure 5.1: NetBeans Text Editor View and Running the Main Project

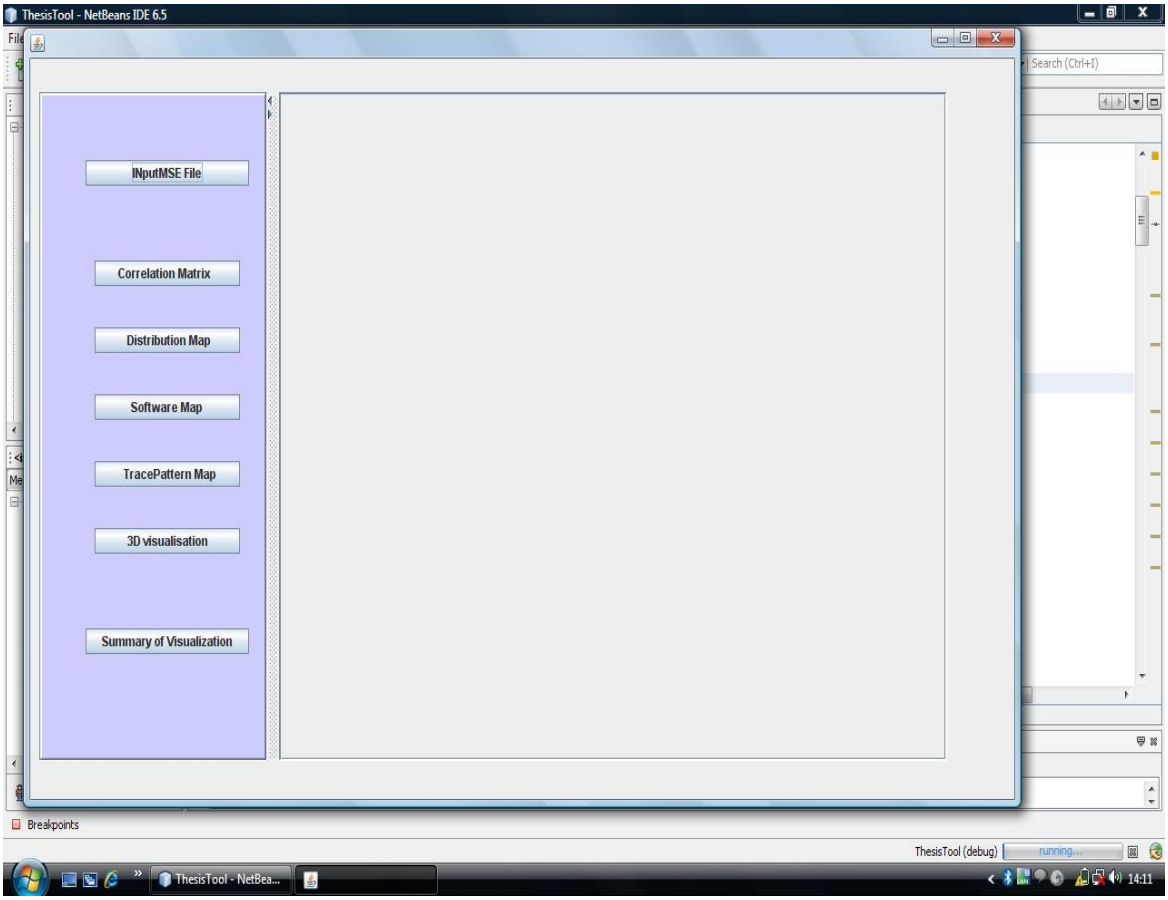

Figure 5.2: Initial Screen shot of the tool.

5.1 Preprocessing and Correlation Matrix

First, pre-process the source code to obtain a term document matrix. The input data is the source code, broken into pieces at an arbitrary level of granularity (e.g., modules, classes, methods etc.) to define the documents used by LSI. The terms are all words found in the source, except keywords of the programming language. LSI is then applied to the term-document matrix to build an index with semantic relationships. From this index we can compute the semantic similarity between both software artifacts and terms and build a correlation Matrix. The whole process is being stored as a .mse file format (Moorse Software reengineering format), by processing this data, obtain the correlation Matrix given in Figure 5.3. Before that selecting the appropriate .mse file, here jEdit (Java Text Editor) case study is being used. Figure 5.2 shows the selection of .mse file of jEdit.

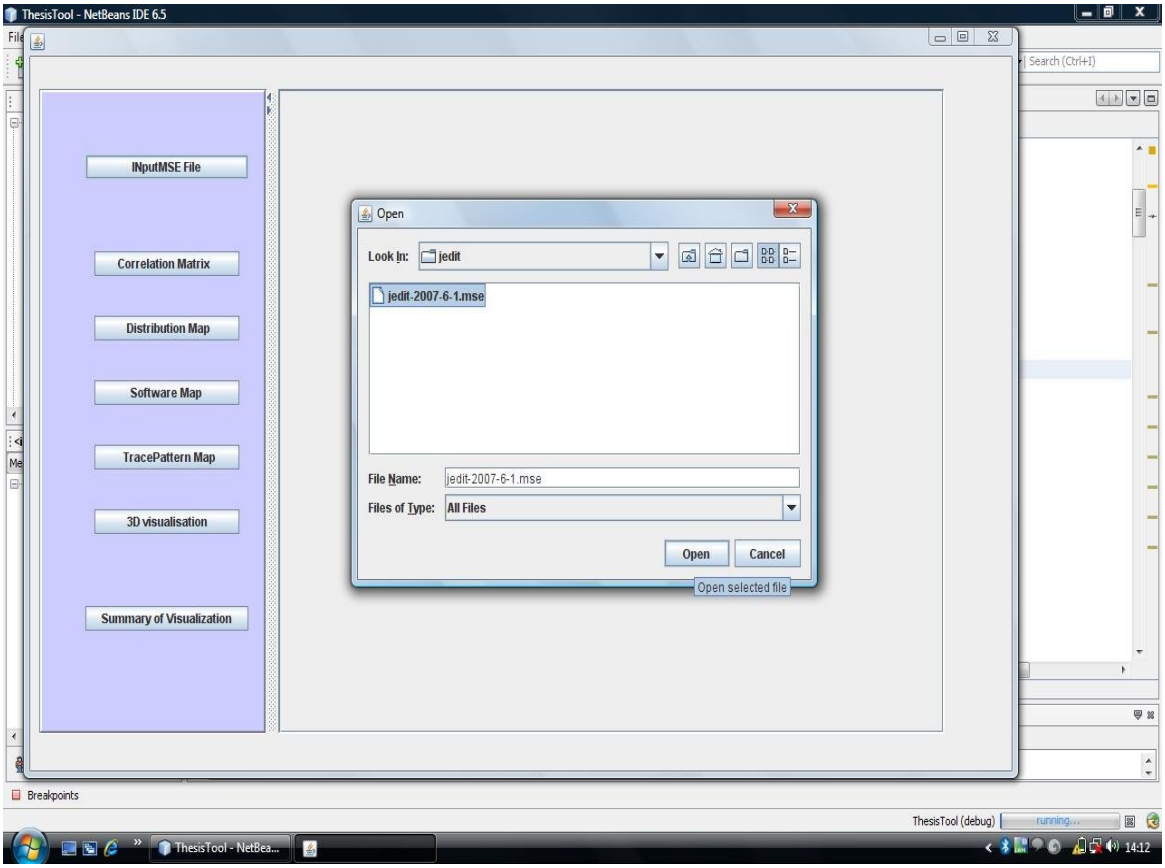

Figure 5.3: Screen Shot showing the Selection of the Input File

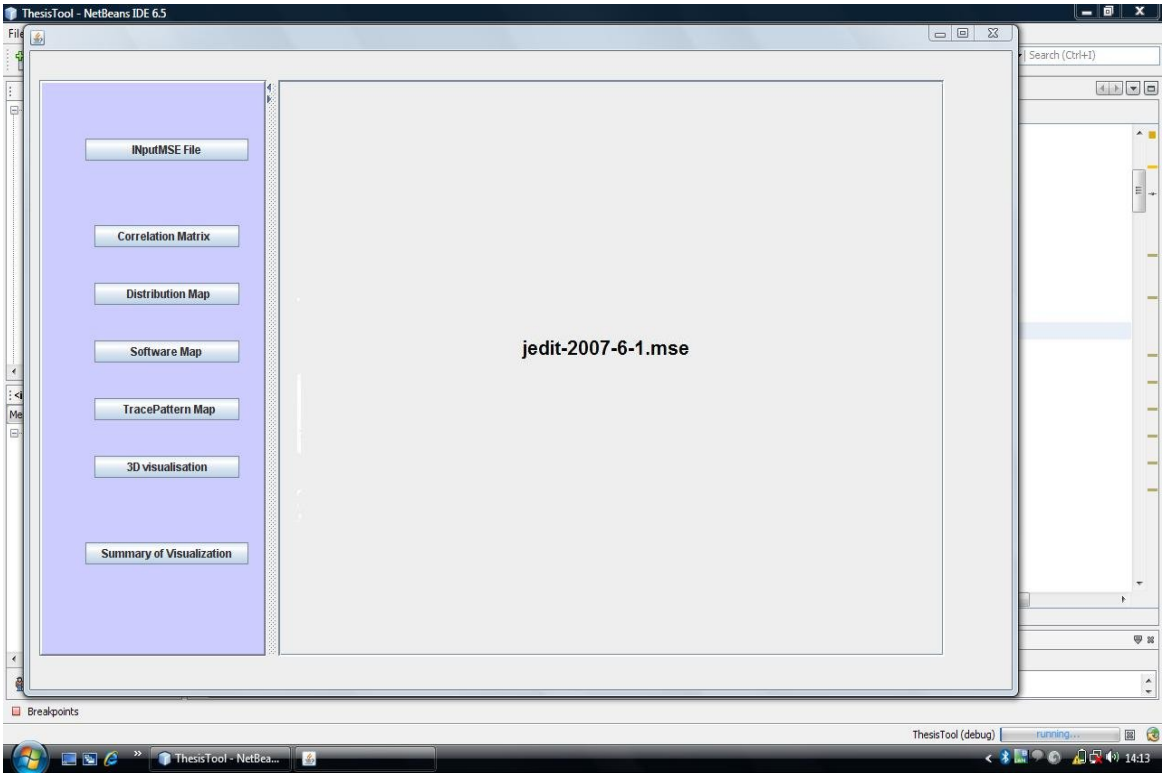

Figure 5.4: Screen Shot of Algorithm running on Selected File

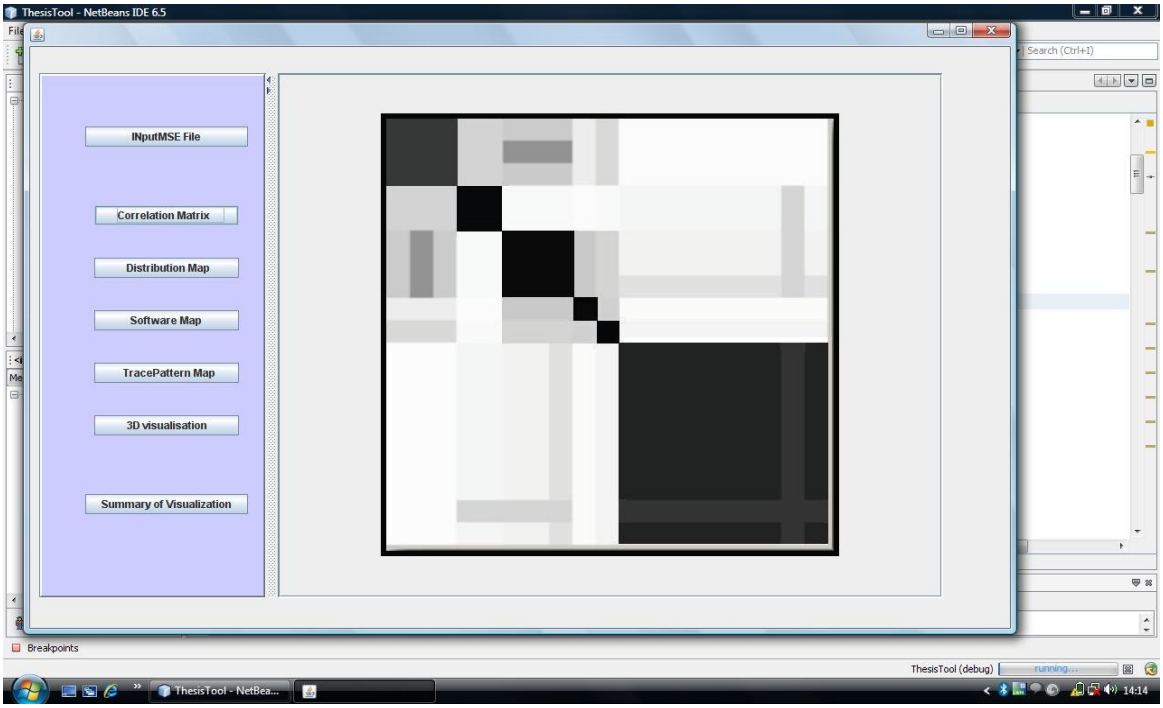

Figure 5.5: Correlation Matrix for jEdit

5.2 Distribution Map

The correlation between the semantic clustering and the structural modularization of a system is represented by Distribution Map. JEdit is a text editor written in Java, the source has a total of 394 classes in 31 packages and uses a vocabulary of 1603 distinct terms and applying semantic clustering resulted in nine domain concepts. Figure 5.6 illustrates how the retrieved domain concepts are distributed over the package structure: the parts are the packages, the elements are the classes and the colours refer to the detected clusters. The table below lists for each concept its size and the description of concept.

Colour	Size	Concept
Red	116	(main domain Concept)
Blue	80	BeanShell scripting
Cyan	68	Regular expressions
Green	63	User Interface
Pink	26	Text Buffers
Dark-green	12	TAR and ZIP archives
Yellow	10	Dockable windows
Magenta	10	XML reader
Orange	9	Bytecode assembler

Table 5.1: jEdit Distribution Map Properties

The concepts description created using the following process:

- For well-encapsulated concepts, the description has been derived from the package name are depicted in **Blue** colour.
- In the case of cross-cutting concepts, the description has been derived from the labels automatically retrieved by Semantic Clustering, which are basically a top-ten list with the most related terms are indicated **Yellow or Pink** colour.
- The labels for the **dark-green** cluster are: curr, buff, idx, archive, TAR, rcdSize, blkSize, rec, heap and ZIP – thus we assigned the concept "TAR and ZIP archives" to this cluster.
- In Figure 5.6 the distribution of **Red**, the largest cluster and thus the main domain concept of the application, shows which parts of the system belong to the core and which do not. Based on the ordering of the packages, we can conclude that the two UI concepts, Green and Yellow, are more closely related to the core.
- The three well-encapsulated concepts **(Orange, Blue and Cyan)** implement clearly separated concepts such as scripting and regular expressions.
- The concepts with the lowest encapsulation cross-cut the system: **Yellow** implements dockable windows, a custom GUI-feature, and **Pink** is about handling text buffers. These two concepts are good candidates for a closer inspection, since we might want to refractor them into packages of their own.

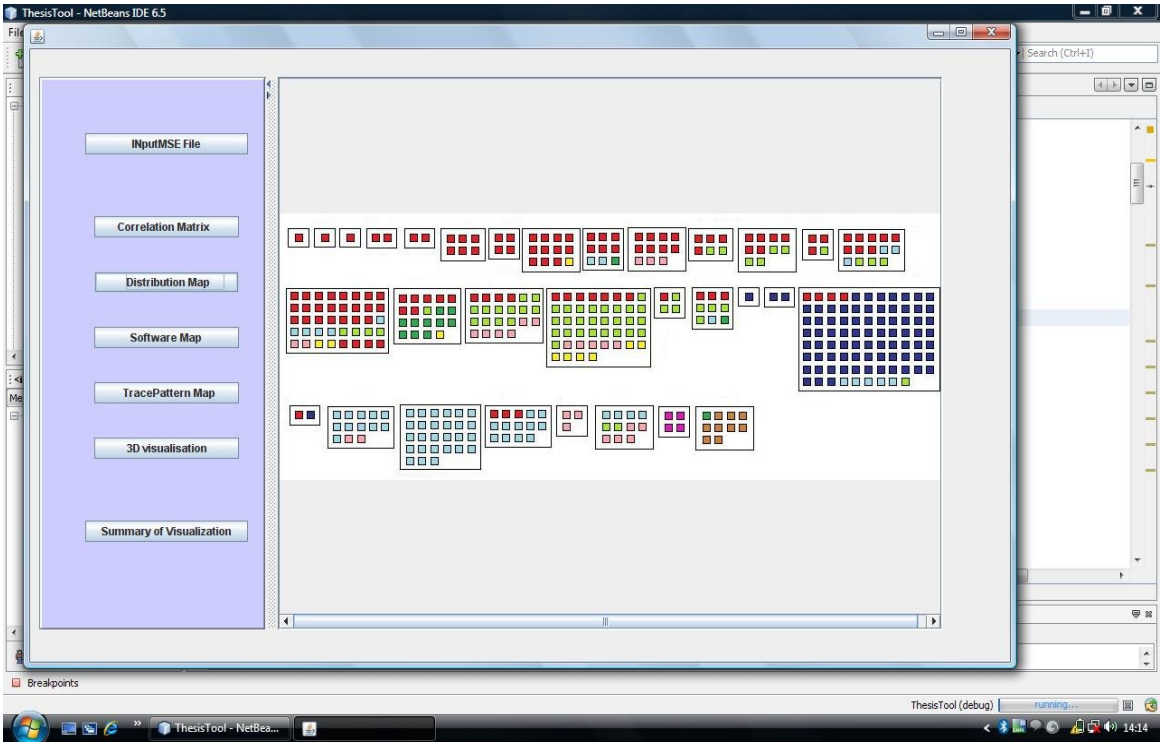

Figure 5.6: Distribution Map

5.3 Software Map

In software visualization, the position of software artifacts and the distance between them is required to reflect a natural notion of position and distance in reality. Instead of looking for distance metrics in the graph structure of programs, focus has been made on the vocabulary of source code artifacts as the space within which their position and distance are defined. Lexical similarity denotes how close software artifacts are in terms of their source code's vocabulary. The vocabulary of the source code corresponds to the implemented technical or domain concepts. Artifacts with similar vocabulary are thus conceptually and topically close [5].

The concept used to draw the software map is multidimensional scaling. Multidimensional Scaling (MDS) tries to minimize a stress function while iteratively placing elements into a low-level space. MDS yields the best approximation of a vector space's orientation, i.e., preserves the relation between elements as best as possible. Thus, MDS is the best choice for mapping the LSI vector space to our target visualization space. MDS attempts to arrange objects in a low-dimensional space so that the distance between them in the target space reflects their similarity. MDS is an iterative algorithm. Given as an input the similarity between objects, it works as follows:

- Assign all objects an arbitrary location in the solution space.
- Determinate the goodness of fit, i.e., compare the distance between the objects in the solution space with their similarities given in the input.
- If the stress value, i.e., the goodness of fit, is within a given threshold, terminate.
- Search for a monotonic transformation of the data. That is, far apart but similar objects are moved towards each other, close but not similar objects are moved away from each other. Proceed with step 3.

During the second step it is important to have a good measure for the goodness of the approximation. A stress value of 0 stands for an optimal solution where the distances between the objects in the configuration perfectly fit their dissimilarity. A higher stress value indicates an increased approximation level between distances and dissimilarities.

The hill-shading algorithm is well-known in geographic visualization. It adds hill shades to a map [33]. The algorithm works on a distinct height model (digital elevation model) rather than on trigonometric data vector date: each pixel has an assigned z-value, its height.

The digital elevation model is a simple matrix with discrete height information for all pixels of the visualization plane. Each element (i.e. source file of class) is represented by the a hill which corresponds to the element's height. The shape of the hill is determined using a normal distribution function. Drawing contour lines on maps is a very common technique. Contour lines make elevation more evident than hill-shading alone. A map without labels is of little use. On a software map, all entities are labeled with their name (class or file name). Figure 5.7 shows the software map for jEdit:

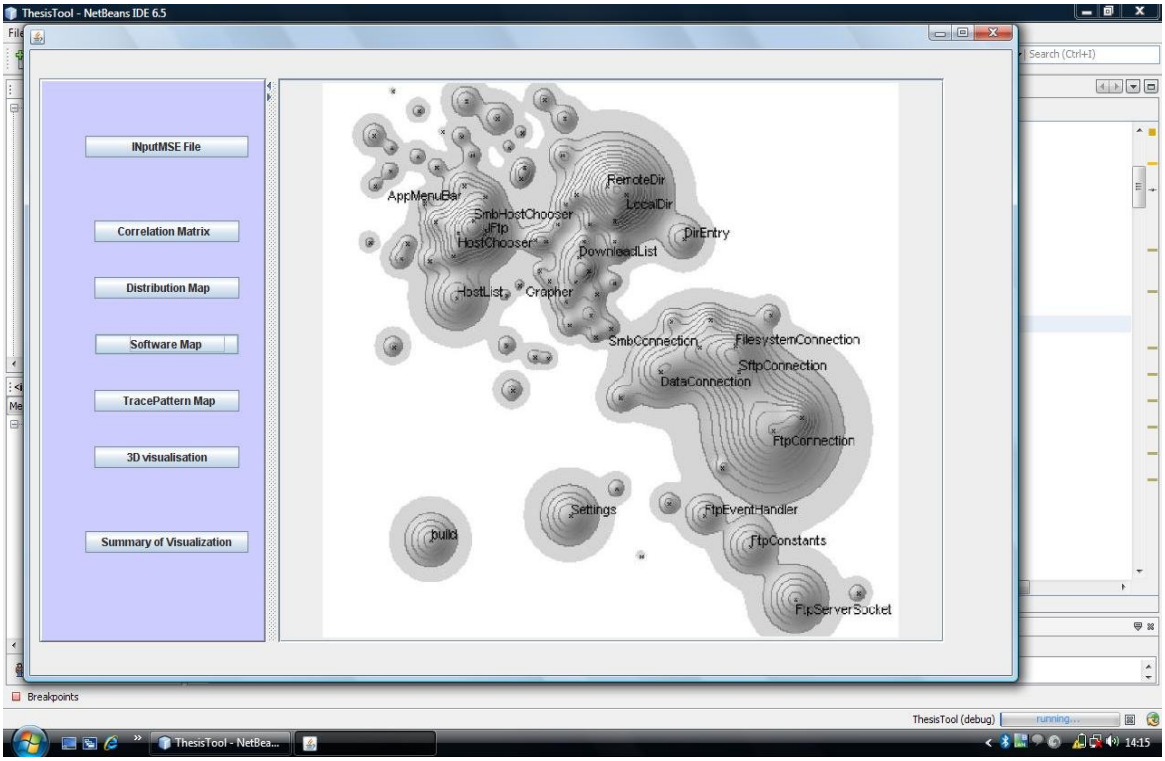

Figure 5.7: Software Map for jEdit

5.4 Trace Pattern Visualization

Traces are the frequency of the word occurrence in the term-document matrix. The traces can be converted into signals by time plot of occurrence of words. As any chronological sequence of partially ordered data points qualifies as a signal, and can draw analogy between traces and signal processing. A key benefit of treating traces as signal is that we get time plot for free. A trace signal is composed of monotone subsequences separated by point wise discontinuities. Patterns visualizations are checked to identify the similar concept in the system. The approximate similar patters correspond to the labelling of particular package or module or class of the system. Figure 5.8 shows the trace patterns of

one of the package of jEdit, in which similar patterns are being visualized by bold colours:

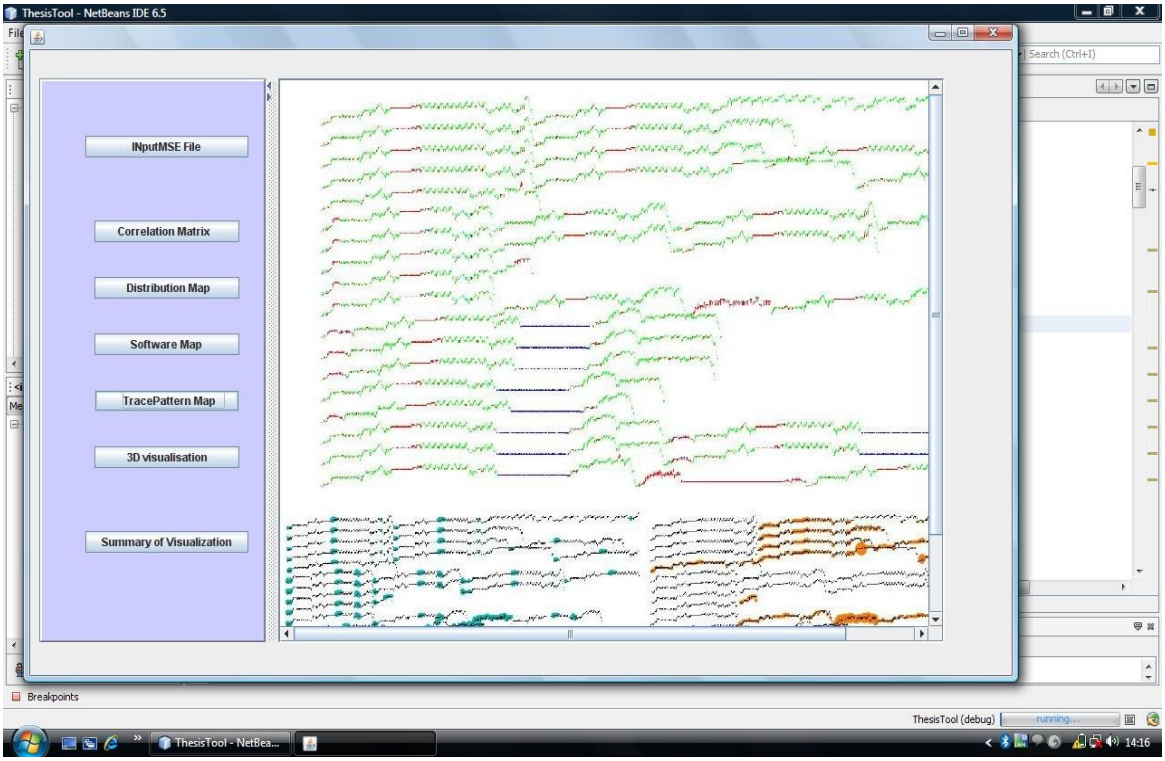

Figure 5.8: Visualization of Trace Patterns

5.5 3-Dimensions Visualization of Software System

Software visualization provides useful assistance in tasks related to program comprehension and reverse engineering, because humans are good at spotting patterns. Software visualization in 3D allows for data exploration in environment closer to the ones we live in. Code City is a 3D software visualization tool based on a city metaphor. It provides assistance in exploring software systems. To allow access to the configuration mechanism, a graphical user interface (GUI) is provided. Here we have used city metaphor for our visualizations because a city is an intuitive exploratory environment with a clear notion of locality. In code city, the classes are represented as buildings and the packages as districts. Some of the visual properties of the city artifacts carry information about the software element they represent. In code city visualization of software system, most of the building colour is blue, their height represents the number of methods metric of the class and the base size the number of attribute metric. The colour of the districts depicts the nesting level of the package they represent, according to the colour scheme, e.g. ranging from dark gray for root package to light gray deeply nested

packages. The clusters visualization can also be visualized through the view configuration mechanism.

A view configuration is a specification defining for each model element type (e.g. class, package, method etc.):

- The visibility, a Boolean denoting whether it will be depicted or not.
- The associated glyph type (what 3D construct is used for representation)
- The layout to use when placing its components
- The visual mappers associated with each property of the chosen glyph.

The user access the configuration mechanism by means of a view configuration user interface (given in Figure 5.9), which provides facility for the modification of every view configuration parameter.

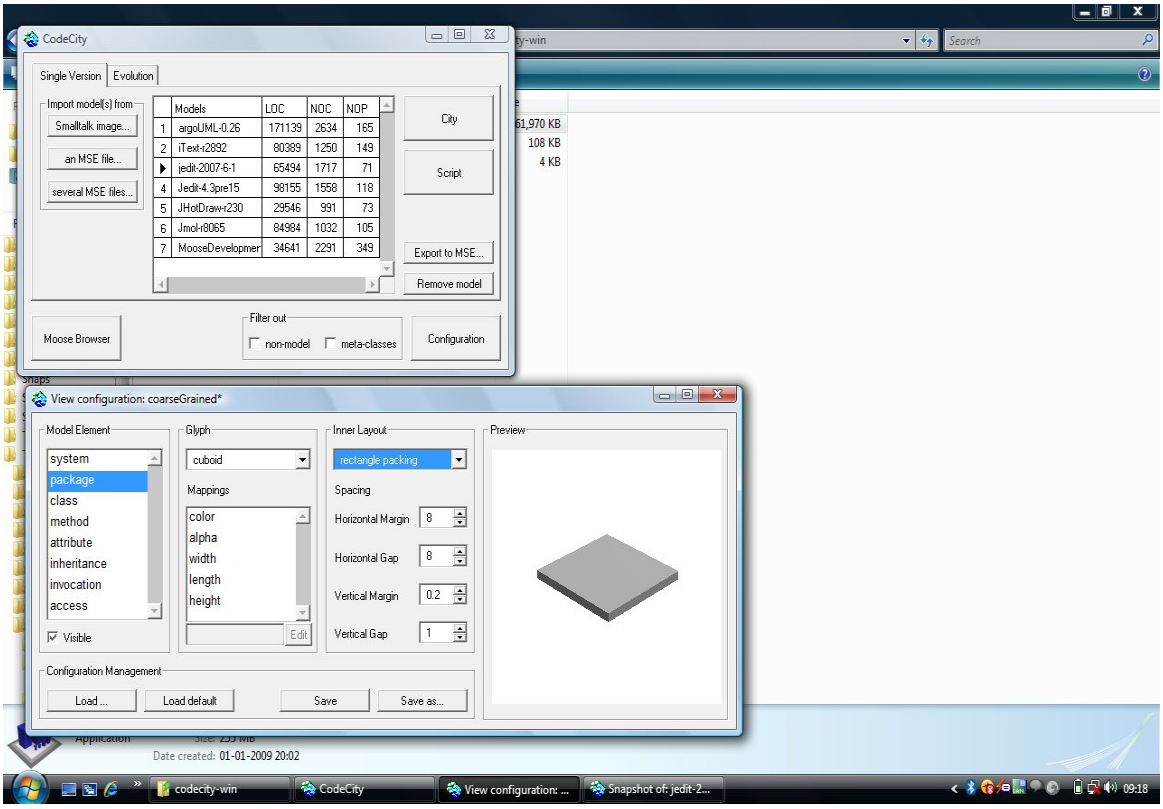

Figure 5.9: View of configuration user Interface of Code City

The Code City visualization window is made of the interactive visualization and an information panel, on the right side, which shows details on the focused element. The height of the building corresponds to the base parameter for the detection of the concept from the system. The code city visualization of the jEdit and our own implementation for the city visualization is given in Figure 5.10 and Figure 5.11 respectively:

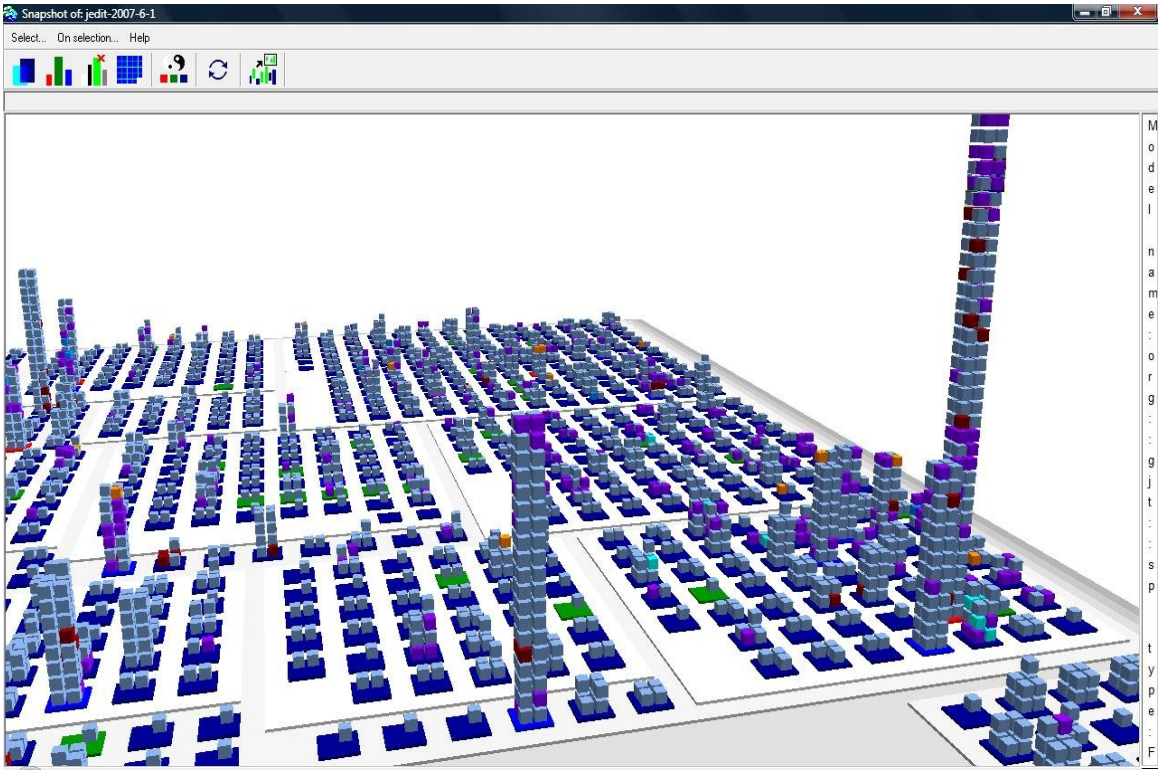

Figure 5.10: Code City Visualization of Clusters

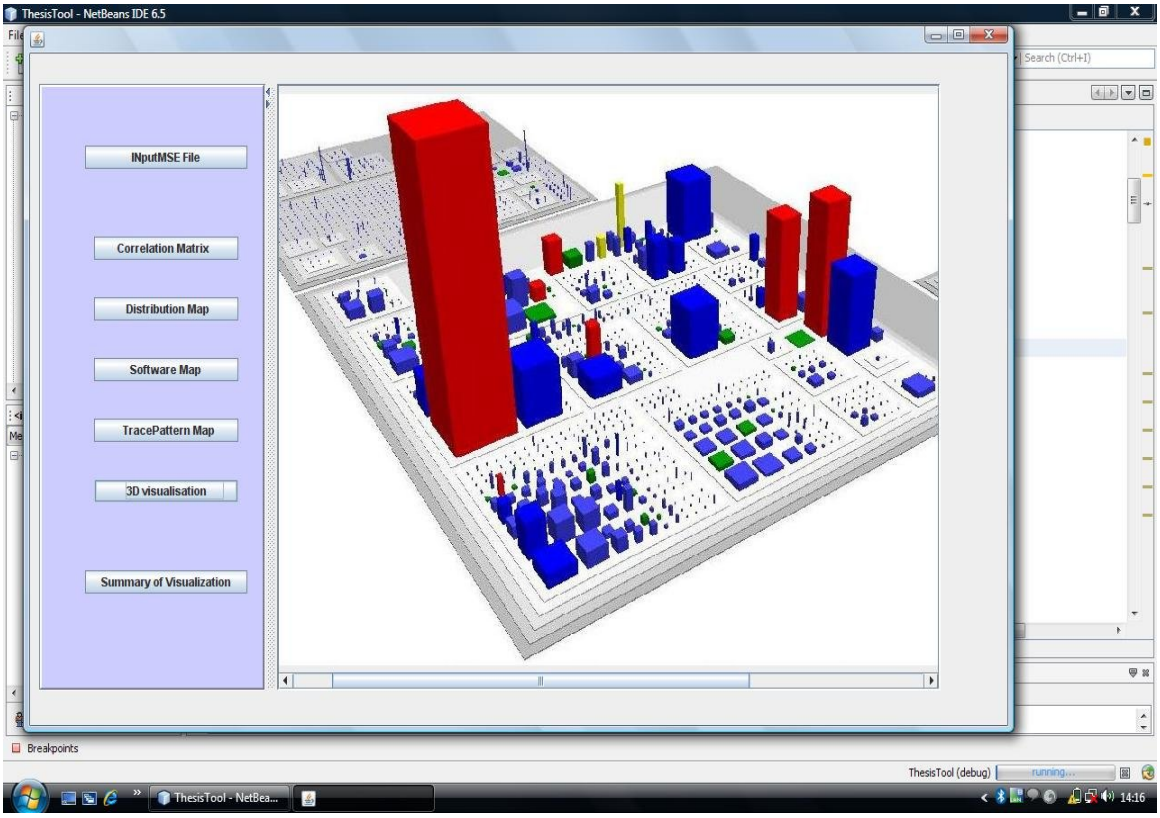

Figure 5.11: Visualization of System as City metaphor

5.6 RESULTS

Code City visualizes the system as city metaphor which shows all the artifacts in the 3-Dimensional visualization. There are lots of parameters to be consider for the concept location, most appropriate in 3D is the height of the buildings as the classes, which composed of the large text corpus. As the height of the building varies with the corpus size, the topic identification is much simpler. The table of jEdit classes (below table 5.1) with their corresponding height parameter and the respective result for the topic identification is given as in Figure 5.12 below:

S.No.	Class Name	Height
1	AbstractOption	2.5
2	ViewConfig	1.1
3	Properties	14.2
4	MiscUtilities	5
5	TextArea	22.7
6	EditBuffer	11
7	SearchAndReplace	3.7
8	JEditBuffer	12.3
9	TextAreaPainter	6.4
10	View	15
11	WorkThreadPool	2.2
12	Log	1.7
13	OptionDialog	3
14	GUIUtilities	8.2
15	AbstractInputHandler	0.9

Table 5.2: Code City Evaluation

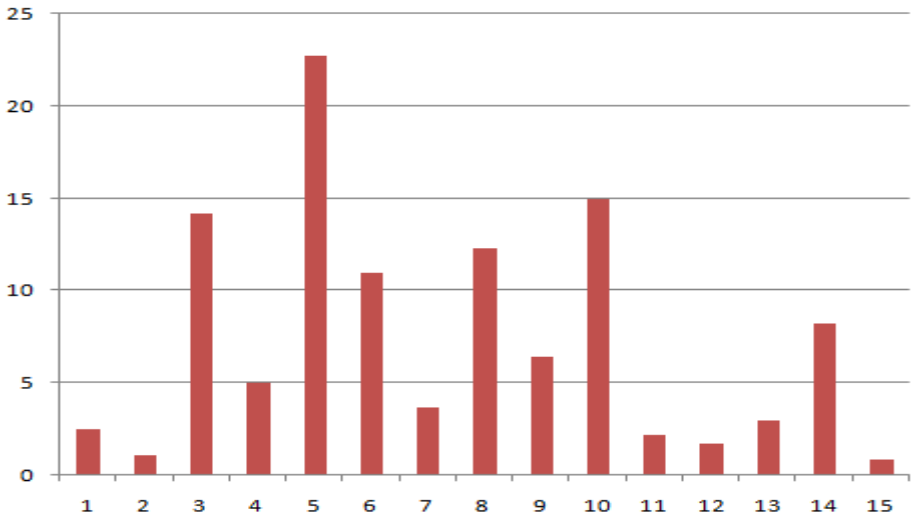

Figure 5.13: Graphical Representation of Detected Concepts in jEdit

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

When understanding a software system, analyzing its structure reveals only half of the story. The other half resides in the domain semantics of the implementation. Many reverse engineering approaches focus only on structural information and ignore semantic information like the naming of identifiers or comments. But developers put their domain knowledge into this part of source code.

This work presented here uses Semantic Clustering to analyze the textual content of source code to recover domain concepts from the code itself. To identify the different concepts in the code, we applied Latent Semantic Indexing (LSI) and the vector space is obtained, based on which we can compute the similarity between either documents or terms.

We clustered the source artifacts according to the vocabulary of identifiers and comments. Since each cluster represents a distinct domain concept, LSI is used as a search engine to retrieve the most relevant labels for the cluster. The labels are obtained for each cluster by ranking and filtering the most similar terms.

Then we obtained the shaded correlation matrix to visualize the identified concept. It elaborates the semantic similarity between source artifacts and the partition of the system into groups of software artifacts that implement the same domain concept, and also shows the semantic links between these clusters, that is single software artifacts which interconnect the main domain concepts.

Visualization is shown based on the concept of distribution map, which shows the structure of the system and how the concept is distributed on the system. The software map visualization shows the concept similarity and distribution on the bases of distance and position. Trace pattern is also one of the techniques to represent the concept detection process to show the patterns similarity using the patterns in the form of traces and signals. Visualization of the clustering in 3-dimension is our basic interest for the domain detection process. Software system shown in the form of city metaphor and the concept is located through the height of the building, which is semantically clustered.

We implemented a tool for the clustering of similar patterns along with Hapax [31] and Code City [32] tools, which are built on top of Moose reengineering environment.

6.2 Future Scope of Work

The major achievements of our work have been compiled and listed in Chapter 5, but there is still a lot of scope for improvement. The proposed work is detecting domain from source code using semantic clustering. The given work can be extended further:

- To investigate in more depth the relationship between the concepts and structure. We would like to compare the results of the semantic clustering with other types of clustering and will also improve the labelling with other techniques.
- To recover the architecture of the system through the semantic clustering technique.
- Can integrate LSI engine in the NetBeans IDE to support documentation and search.

References

[1] A. Abran, P. Bourque, R. Dupuis, L. Tripp, “Guide to the software engineering body of knowledge (ironman version)”, Tech. rep., IEEE Computer Society (2004).

[2] S. Ducasse, M. Lanza, “The class blueprint: Visually supporting the understanding of classes”, IEEE Transactions on Software Engineering 31 (1) (2005) 75–90.

[3] Y. S. Maarek, D. M. Berry, G. E. Kaiser, “An information retrieval approach for automatically constructing software libraries”, IEEE Transactions on Software Engineering 17 (8) (1991) 800–813.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, “Recovering traceability links between code and documentation”, IEEE Transactions on Software Engineering 28 (10) (2002) 970–983.

[5] Adrian Kuhn, Stephane Ducasse, Tudor Girba, “Semantic Clustering: Identifying Topics in Source Code”, Language and Software Evolution Group, LISTIC, Universite de Savoie, France, 2006

[6] Yo¨elle S. Maarek, Daniel M. Berry, and Gail E. Kaiser, “An information retrieval approach for automatically constructing software libraries”, IEEE Transactions on Software Engineering, 17(8):800–813, August 1991.

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo, “Recovering traceability links between code and documentation”, IEEE Transactions on Software Engineering, 28(10):970–983, 2002.

[8] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, R. A. Harshman, “Indexing by latent semantic analysis”, Journal of the American Society of Information Science 41 (6) (1990) 391–407.

[9] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, “An information retrieval approach to concept location in source code”, in: Proceedings of the 11thWorking Conference on Reverse Engineering (WCRE 2004), 2004, pp. 214–223.

[10] S. T. Dumais, J. Nielsen, “Automating the assignment of submitted manuscripts to reviewers, in: Research and Development in Information Retrieval”, 1992, pp. 233–244.

[11] J. I. Maletic, A. Marcus, “Using latent semantic analysis to identify similarities in source code to support program understanding”, in: Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000), 2000, pp. 46–53.

[12] S. Kawaguchi, P. K. Garg, M. Matsushita, K. Inoue, Mudablue, “An automatic categorization system for open source repositories”, in: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), 2004, pp. 184–193.

[13] A. Marcus, J. I. Maletic, “Identification of high-level concept clones in source code”, in: Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001), 2001, pp. 107–114.

[14] A. De Lucia, F. Fasano, R. Oliveto, G. Tortora, “Enhancing an artefact management system with traceability recovery features”, in: Proceedings of 20th IEEE International Conference on Software Maintainance (ICSM 2004), 2004, pp. 306–315.

[15] A. Marcus, D. Poshyvanyk, “The conceptual cohesion of classes”, in: Proceedings Internationl Conference on Software Maintenance (ICSM 2005), IEEE Computer Society Press, Los Alamitos CA, 2005, pp. 133–142.

[16] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue, “Mudablue: An automatic categorization system for open source repositories”, In Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC.04), pages 184–193, 2004.

[17] Bruno Caprile and Paolo Tonella. Nomen est omen, “Analyzing the language of function identifiers”, In Proceedings of 6th Working Conference on Reverse Engineering (WCRE 1999), pages 112–122. IEEE Computer Society Press, 1999.

[18] Nicolas Anquetil and Timothy Lethbridg, “Extracting concepts from file names; a new file clustering criterion”, In International Conference on Software Engineering (ICSE’98), pages 84–93, 1998.

[19] Jaques Bertin, “Graphics and Graphic Information Processing”, Walter de Gruyter, 1981.

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

[21] Michael W. Berry, Susan T. Dumais, and Gavin W. O’Brien, “Using linear algebra for intelligent information retrieval”, SIAM Review, 37(4):573–597, 1995

[22] Adrian Kuhn, St´ephane Ducasse, and Tudor Gˆırba, “Enriching reverse engineering with semantic clustering”, In Proceedings of Working Conference on Reverse Engineering (WCRE 2005), pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[23] Adrian Kuhn, Stephane Ducasse, and Tudor Girba, “Semantic clustering: Exploiting source code linguistic information”, Information and Software Technology, submitted, 2006.

[24] Adrin Kuhn, Orla Greevy, and Tudor Girba, “Applying semantic analysis to feature execution traces”, In Proceedings of Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005), pages 48-53, November 2005.

[25] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt, “Comparison of clustering algorithms in the context of software evolution”, In Proceedings of International Conference on Software Maintenance, pages 525-535, 2005.

[26] Stephane Ducasse, Tudor Girba, and Adrian Kuhn, “Analyzing the distribution of properties in software systems”, In Submitted to International Conference on Program Comprehension (ICPC 2006), 2006.

[27] Salton, G. and McGill, M., “Introduction to Modern Information Retrival”, McGraw-Hill, 1983.
[28] H. P. Luhn, “The automatic creation of literature abstracts”, IBM Journal of Research and Development, 2:159-165, 1958.
[29] George Kingsley Zipf, “Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology”, Addison-Wesley Press Inc., Cambridge 42, MA, USA, 1949.
[30] Chris Buckley, “Implementation of the smart information retrieval system”, Technical Report TR85-686, Cornell University, Ithaca, NY, USA, 1985.
[31] Adrian Kuhn, “Hapax – enriching reverse engineering with semantic clustering”, November 2005.
[32] Richard Wettel, “Scripting 3D visualizations with Code City”, 2008.
[33] Terry A. Slocum, Robert B. McMaster, Fritz C. Kessler, and Hugh H. Howard, “Thematic Carthography and Geographic Visualization”, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2005.
[34] Martin F. Porter, “An algorithm for suffix stripping”, Program, 14(3):130-137, 1980.
[35] Susan T. Dumais, “Improving the retrieval of information from external sources, Behaviour Research Methods, Instruments and Computers”, 23:229-236, 1991.

Appendix

Overview of LSI

Latent Semantic Indexing (LSI) is a machine-learning model that induces representations of the meaning of words by analyzing the relation between words and passages in large bodies of text. LSI has been used in applied settings with a high degree of success in areas like automatic essay grading and automatic tutoring to improve summarization skills in children. As a model, LSI's most impressive achievements have been in human language acquisition simulations and in modeling of high-level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity. LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with vector space model (VSM) [27] approaches. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix decomposed using Singular Value Decomposition. It has been shown in [20] that LSI addresses the synonyms very well. With simple corpus training, LSI managed to answer correctly 64% of the synonyms questions in the Test of English as a Foreign Language, better than the average student.

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

two documents are considered similar if their corresponding vectors in the VSM space point in the same (general) direction.

LSI relies on a Single Value Decomposition (SVD) [27] of the co-occurrence matrix. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a feature space without serious loss of specificity. The formalism behind SVD is rather complex and lengthy to be presented here. One of the most successful applications of SVD in information retrieval is the Google search engine (www.google.com).

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

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

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

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

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

LSI is mostly used on natural language corpora. However, the method lends itself perfectly to other type of data. One criticism of this type of method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. Very good representations and results are derived without this information. This characteristic is well suited to the domain of source code and internal documentation. Source code is hardly English prose but with selective naming, much of the high level meaning of the problem-at-hand is conveyed to the reader. Internal source code documentation is also commonly written in a subset of English so queries formulated in natural language are perfectly usable. This makes automation drastically easier and directly supports programmer defined variable names that have implied meanings (e.g., avg) yet are not in the natural language vocabulary [9].

LIST OF PUBLICATIONS

- Sanjay Madan and Shalini Batra, "*Detecting Domain from Source Code using Semantic Clustering*" published in International Conference on Intelligent Systems and Networks (IISN-2009) held at Institute of Science and Technology, Klawad, Yamuna Nagar from February 14-16, 2009.
- Sanjay Madan and Shalini Batra, "Visualizing Domain in 3-Dimension using Semantic Clustering" published in CiiT International Journal of Data Mining and Knowledge Engineering, June 2009.