Semantic Clustering

Making Use of Linguistic Information to Reveal Concepts in Source Code

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Adrian Ivo Kuhn

im März 2006

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Semantic Clustering

Making Use of Linguistic Information to Reveal Concepts in Source Code

Adrian Kuhn

Abstract

Many approaches have been developed to comprehend software source code, most of them focusing on program structural information. However, in doing so we are missing a crucial information, namely, the *domain semantics* information contained in the text or symbols of the source code. When we are to understand software as a whole, we need to enrich these approaches with conceptual insights gained from the domain semantics. This paper proposes the use of information retrieval techniques to exploit linguistic information, such as identifier names and comments in source code, to gain insights into how the domain is mapped to the code. We introduce *Semantic Clustering*, an algorithm to group source artifacts based on how they use similar terms. The algorithm uses Latent Semantic Indexing. After detecting the clusters, we provide an automatic labeling and then we visually explore how the clusters are spread over the system. Our approach works at the source code textual level which makes it language independent. Nevertheless, we correlate the semantics with structural information and we apply it at different levels of abstraction (for example packages, classes, methods). To validate our approach we applied it on several case studies.

Table of Contents

1 Introduction **1**
- 1.1 Semantic Clustering . . . 3
- 1.2 Structure of the Thesis . . . 4

2 Latent Semantic Indexing **6**
- 2.1 Overview . . . 6
- 2.2 Vectorial Semantics . . . 7
- 2.3 Data Normalization . . . 8
- 2.4 Singular Value Decomposition . . . 9
- 2.5 Term and Documents Similarity . . . 11
- 2.6 Searching for Terms or Documents . . . 12

3 Taking Software Analysis Beyond Mere Structure **14**
- 3.1 Linguistic Information . . . 15
- 3.2 Software Reuse . . . 15
- 3.3 Traceability Recovery . . . 17
- 3.4 Feature Location . . . 19
- 3.5 Other Application of LSI . . . 20
- 3.6 Software Clustering . . . 20

4 Semantic Clustering **23**
- 4.1 Building the Text Corpus . . . 24
- 4.2 Semantic Similarity . . . 25
- 4.3 Semantic Clustering . . . 25
- 4.4 Semantic Links . . . 27
- 4.5 Correlation Matrix . . . 28
- 4.6 Labeling the Clusters . . . 29

5 Distribution of Concepts **31**
- 5.1 The Distribution Map . . . 32

5.2 Distribution Patterns 33
5.3 Case-Study 34

6 Experiments **38**
6.1 Semantic Links between a Framework and its Plug-ins . . 39
6.2 Semantic Clusters of Methods in a Class 42
6.3 Distribution of Semantic Clusters in JBoss 44
6.4 Distribution of Semantic Clusters in Azureus 45
6.5 Distribution of Semantic Clusters in JEdit 47
6.6 Distribution of Semantic Clusters in Ant 49

7 Discussion **51**

8 Conclusions **56**

List of Figures **59**

Bibliography **61**

Chapter 1

Introduction

"What's in a name? That which we call a rose
By any other word would smell as sweet."
— Shakespeare

Many approaches have been developed to comprehend software systems, most of them focusing on structural information. Other approaches using methods such as dynamic data or history information have also proven valuable. However, if we are to understand software as a whole we need to enrich these approaches with conceptual insight into the domain semantics of the software system.

> *The informal linguistic information that the software engineer deals with is not simply supplemental information that can be ignored because automated tools do not use it. Rather, this information is fundamental. [...] If we are to use this informal information in design recovery tools, we must propose a form for it, suggest how that form relates to the formal information captured in program source code or in formal specifications, and propose a set of operations on these structures that implements the design recovery process* [Big89].

This work proposes to use a combination of information retrieval (IR) and clustering to exploit linguistic information, such as identifier names and comments, to gain insight into the domain semantics of a software system.

The underlying assumption is that developers use meaningful names for code items, for example names that convey their domain knowledge.

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.

Source code *is* a twofold means of communication[1]: on the formal level of syntax it is communication between developer and machine, but – much more important – on an informal level it is about communication between developers. Code is not written for machines – that is what executables are for – code is written for humans. Consider, for example, a short testing method telling 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;
}
```

Stripping away all identifiers and comments, the functionality remains the same, but the meaning becomes obfuscated and almost impossible to tell. In our example, removing informal information yields:

```
public type_1 method_1(type_2 a, type_2 b, type_2 c) {
    if (!method˙2(a, b ,c)) throw Exception(literal_1).
    return (a op_1 A) op_2 (b op_1 B) op_2 (c op_1 C);
}
```

On the other hand, retaining only the informal information yields the following. And even though the vocabulary is presented in random order, the domain of the method is still recognizable without doubt.

```
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
```

This representation of documents as bag-of-terms is a well-established

[1]Source code also contains communication between developer and user: for example text snippets in the source code displayed to the user. But for the purpose this dissertation, this can be subsumed under developer communication without loss of generality.

technique in information retrieval (IR), and is used to model documents in a text corpus. Information retrieval provides means to analyze, classify and characterize text documents based on their content. There are approaches in the field of software analysis that apply IR on external documentation [MBK91, ACC+02], 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 [CT99]. The main difficulty they found is that source code has a quite scarce and ambiguous vocabulary compared to natural langauge documents. Marcus *et al* proposed Latent Semantic Indexing (LSI), a information retrieval technique that takes synonymy and polysemy into account, as a means to locate concepts [MM00, MSRM04]. Recently Kawaguchi *et al* used LSI and clustering to categorize software projects at the level of entire projects [KGMI04].

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 programm slices like execution traces [KGG05]. Additionally, it introduces two visualizations: one focused on the correlation among semantics and one focused the relation between semantics and structure.

1.1 Semantic Clustering

This work introduces *Semantic Clustering*, a novel technique to characterize 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. It takes software comprehension from a low-level view, lost in the complex structure of thousands of artifacts, to a high-level view that characterizes the system by a handful of abstract domain concepts.

Semantic clustering is a non-interactive and unsupervised technique. 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 results in a search index on software artifacts (more details on LSI see Chapter 2). With that index at hand, the software system

is grouped into clusters. A semantic cluster is a group of artifacts that use the same vocabulary. Therefore each cluster reveals a different concept found in the system.

Finally, the inherently unnamed concepts are named with labels taken from the vocabulary of the source code. An automatic algorithm labels each cluster with is most related terms, and provides in this way a human readable description of the main concepts in a software system.

To present the semantic clustering to the reverse engineer, we employ two visualizations. The *Correlation Matrix* illustrates the relation among concepts. The *Distribution Map* illustrates the between concepts and the structure of the system.

We implemented this approach in a tool called Hapax[2], which is built on top of the Moose reengineering environment [DGLD05], and we applied it to several case studies.

The contributions of this work are:

- combining the structure of software systems with semantical information [KDG05],
- defining two visualizations to illustrate the result: the correlation matrix [KDG05] and the distribution map [DGK06],
- an automatic labeling of the clusters [KDG05, KDG06],
- interpretation of the system at different level of abstraction [KGG05, LKGL05].

1.2 Structure of the Thesis

Chapter 2 introduces Latent Semantic Indexing (LSI), which is the information retrieval (IR) technique used in this work. It covers the mathematical background, and describes how to measure similarity and how to search for documents. Chapter 3 browses the state-of-the-art in semantic driven software comprehension. In Chapter 4 we show how we use LSI to

[2]The name is derived from the term *hapax legomenon*, which refers to a word occurring only once a given body of text.

analyze the semantics of the system and how we can apply the analysis at different levels of abstraction. In Chapter 5 we show how to visualize the distribution of concepts over the structure, and propose a terminology to describe common distribution patterns. In Chapter 6 we exemplify the results on five case studies emphasizing the different aspects of the approach. We discuss different variation points in Chapter 7, and in Chapter 8 we conclude and present future work.

Chapter 2

Latent Semantic Indexing

"Ach, wie gut ist, da niemand weiß
Daß ich Rumpelstilzchen heiß!"
— Rumpelstilzchen

This chapter covers Latent Semantic Indexing (LSI), a technique common in information retrieval to index, analyze and classify text documents [DDL+90]. LSI 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 sapce and unrelated documents far apart of each other. Since source code is basically composed of text documents as well, we use LSI to analyze the linguistic information of a software system.

2.1 Overview

As most 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.

LSI has been developed to overcome problems with synonymy and polysemy that occurred in prior vectorial approaches, and thus it 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 kind of principal components analysis originally used in signal processing to reduce noise while preserving the original singal. Assuming that the original term-document matrix is noisy (the aforementioned synonymy and polysemy), the approximation is interpreted as a noise reduced – and thus better – model of the text corpus.

As an example: a typical text corpus with millions of documents, containing some ten thousands of terms, is reduced to a vector space with 200-500 dimensions only.

Even though search engines are the most common usage of LSI [BDO95], there is a wide range of applications, such as: automatic essay grading [FLL99], automatic assignment of reviewers to submitted conference papers [DN92], cross-language search engines, thesauri, spell checkers and many more. Furthermore LSI has proved useful in psychology to simulate language understanding of the human brain, including processes such as the language acquisition of children and other high-level comprehension phenomena [LD91].

In the field of software engineering LSI has been successfully applied to: categorized source files [MM00] and open-source projects [KGMI04], detect high-level conceptual clones [MM01], recover links between external documentation and source code [LFOT04, MP05] and to compute class cohesion [MP05]. See Chapter 3 for more details.

2.2 Vectorial Semantics

The vector space model dates back to Luhn, who observed that *"the frequency of word occurrence in a document furnishes a useful measurement of its content"* [Luh58]. Vectorial approaches represent documents as bag-of-words and arrange them in a Term-Document Matrix A, where $a_{i,j}$ is the number of times term t_i appears in document d_j.

This bag-of-words representation summarizes documents by their term histograms, ignoring the ordering of terms or any other collocation infor-

mation. An advantage of this reduction is that it is easily automated and needs minimal intervention beyond filtering of the term list. In Section 2.4 is explained how LSI enhances this basic model using noise reduction.

The name "vector space model" is used due to the geometrically interpretation of the term-document-matrix. From a geometrical point of view, the columns of the matrix A are vectors representing the documents in an n-dimensional term space. Two documents are considered similar if their corresponding vectors point in the same direction, and the similarity is defined as the cosine or inner product between the corresponding vectors (see Section 2.5).

2.3 Data Normalization

Not all words are 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 [Luh58]. 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 [Zip49], 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* [Buc85]. 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 [Por80].

Research in information retrieval (IR) has shown that normalization leads to more effective retrieval than if the raw term frequencies were used [Dum91]. 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.

$$x_{i,j} = \text{local}(t_i, d_j) \times \text{global}(t_i)$$

When applied on textual data LSI achieves best results with the *entropy* weighting [Nak01]. Nevertheless we present here the *tf-idf*-weighting as it is more popular in both information retrieval and software analysis (see Section 4.1).

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:

$$a_{i,j}^{tfidf} = \log(a_{i,j} + 1) \times \log \frac{|D|}{df_i}$$

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, using the Euclidian norm.

2.4 Singular Value Decomposition

LSI enhances vectorial semantics with singular value decomposition (SVD) to overcome problems with synonymy and polysemy. Synonymy denotes multiple words have the same meaning and polysemy denotes a single word has multiple meanings. This causes problems when using the vector space model (VSM) to build search engines. It can happen that a desired document is missed because the user of a search engine uses a different

word in his query than the author of a document in the document, even though both words have the same meaning. In the same way might the search return unwanted documents due to alternate usages of an ambiguous word.

LSI solves this problems replacing the full term-document matrix A with a low-rank[1] approximation A_k. The 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 (the aforementioned synonymy and polysemy), the approximation A_k is interpreted as a noise reduced – and thus better – model of the text corpus.

$$A = U \times S \times V^{\mathrm{T}}$$

Singular value decomposition transforms the matrix A into three matrices, using eigenvalue decomposition. This yields three matrices: $U \times S \times V^{\mathrm{T}}$. 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, that is the best rank k approximation of A under the least-square-error criterion:

$$A'_{n\times m} = U_{n\times k} \times S_{k\times k} \times V^{\mathrm{T}}_{k\times m}$$

[1] In LSI the rank k of matrix A_k is equivalent the dimension of the corresponding vector space. Thus, in the context of LSI, the terms "dimension" and "rank" are synonymous.

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 [Zip49]. Thus, even if the dimension of matrix A goes into millions, there are typically only about $k = 200 - 500$ relevant singular values. That is why a text corpus with millions of documents can be approximated with such a low-ranked matrix. This is illustrated on Figure 2.1 considering an image, instead of text, as example. If this image would be a text corpus, the columns would be the documents and the lines would be the terms.

Figure 2.1: The same image at different SVD approximations. From top left to bottom right: the original image with rank 200, an approximation with rank 20, rank 10, rank 5, rank 2 and finally rank 1. For the purpose of LSI, the rank 5 approximation would be sufficient.

2.5 Term and Documents Similarity

To interpret the SVD factors geometrically, the rows of the truncated matrices U_k and V_k are taken as coordinates of points representing the documents and terms in a k dimensional space. The nearer one points to the other, the more similar are their corresponding documents or terms (see Figure 2.2). Similarity is typically defined as the cosine between the corresponding vectors:

$$\text{sim}(d_i, d_j) = \cos(\mathbf{v}_i, \mathbf{v}_j)$$

Computing the similarity between document d_i and d_j is done taking the cosine between the i-th and j-th row of the matrix VS, (that is between the vectors of the corresponding points scaled by the singular values). The same goes for terms, but with rows of the matrix $V\Sigma$.

Computing the similarity between term t_i and document d_j differs. It is done taking the cosine between the i-th row of the matrix $U\Sigma^{\frac{1}{2}}$ and the j-row of the matrix $V\Sigma^{\frac{1}{2}}$, that is, between the vectors of the corresponding points scaled by the square-root of the singular values.

Being cosine values, similarity values range from 1 for similar vectors with the same direction 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 stray much below zero.

2.6 Searching for Terms or Documents

Searching is typically done comparing each document with a search query and returns the most similar documents (see Figure 2.2). In the same way, it is possible to search for terms using a set of documents as query. We use this "reverse-search" in Section 4.6 to obtain labels for the concept clusters.

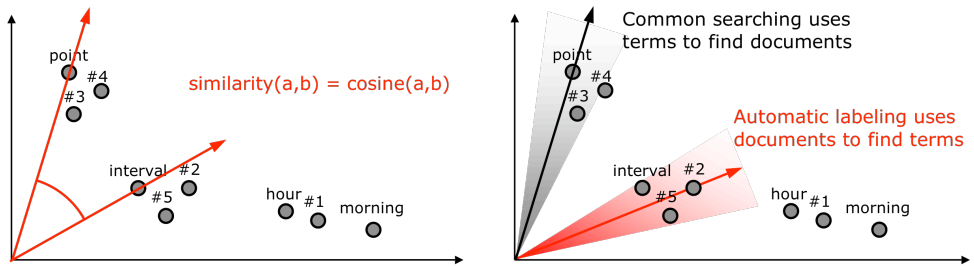

Figure 2.2: On the left: An LSI-Space with terms and documents, similar elements are placed near each other. On the right: the grey cone is a search for documents related to term *point*, the red cone is a reverse-search for terms related to document #2.

A query is a set of words and is thus, in the same way as documents, represented as bag-of-words. This pseudo-document is projected into the k dimensional LSI space as follows:

$$q_k = q \times U_k \times S_k^{-1}$$

where q is a vector with the word occurrences of the search query, normalized in the same way as the original term-document matrix A. This projection places the pseudo-document at the location of the weighted sum of its constituent term vectors. The query is then compared to all existing documents vectors, and the documents are returned in a list ordered by their similarity to the query.

Chapter 3

Taking Software Analysis Beyond Mere Structure

"The primary purpose of the Data statement is to give names to constants. Instead of referring to pi as 3.141592653589793 at every appearance, the variable PI can be given that value. This also simplifies modifying the program, should the value of pi change."
— Fortran manual for Xerox Computers

This dissertation presents an approach using information retrieval (IR) techniques and clustering to analyze the domain semantics of a software system. Early work applying IR dates back to the eighties, however these approaches had been restricted to external documentation written in natural language. But, the last ten years saw two developments that lifted this restriction: First the development of superior IR techniques that are able to cope with the scarce and ambiguous vocabulary of source code, more see Chapter 2 about Latent Semantic Indexing (LSI). And secondly a shift in the culture of software development, moving away from short and abbreviated identifiers towards verbose Naming Conventions using meaningful names (see for example the Java naming convention [Jav]). /AKrephrase

3.1 Linguistic Information

Biggerstaff advocates linguistic information as a being fundamental to software analysis [Big89]. He states that linguistic information is not just supplemental and ignorable, but rather fundamental and thus worth focusing on. As a striking example he removes any linguistic information from a piece of source code, and presents it once without and once with linguistic information to illustrate the difference. Eventually he proposes as one of the major challenges in software analysis tools that deal with linguistic information and relate it to formal information.

The first work that presents techniques to analyze linguistic information is "Nomen est Omen" by Caprile and Tonella. They analyze the lexical, syntactical and semantical structure of function identifiers [CT99]. First a segmentation technique is presented to split identifier names into lexical parts, that is into single words. The segmentation works even if the parts are compound without delimiters such as spaces, underscores or camel case. Then they present an informal grammar for the syntactical structure of identifier names that covers the most common patterns used by developers when choosing identifier names. Formal Concept Analysis (FCA) is applied to build a concepts lattice of the domain semantics of a software systems, which uses the functions as objects and the lexical parts of their names as attributes.

3.2 Software Reuse

The use of information retrieval (IR) techniques in software analysis dates back to the late eighties, originating from the software reuse community. Frakes and Nejmeh proposed to apply information retrieval (IR) on source code as if it would be a natural language text corpus [FN87]. They applied an IR system based on keyword matching, which allowed to perform simple searches using wildcards and set expressions. Back then leading IR technology, this is commonplace nowadays: advanced search abilities such as regular expressions are found in any IDE environment.

Even though Merlo *et al* propose to cluster files according to function names and the content of comments [MMD93], it was not until the current decade that more research into this direction was conducted. All the

approaches presented in this section apply IR on external documentation. This is due to the use of information retrieval (IR) techniques that are less powerful than Latent Semantic Indexing (LSI), and thus not capable of dealing with the vocabulary used in source code.

To automatically detect reusable software libraries Maarek *et al* apply information retrieval (IR) on external documentation [MBK91]. They argue that linguistic information found in source code is too ambiguous and they argue further that it is too complex to relate comments to the portion of code they concern. Their retrieval is based on an improved vector space model (VSM), which does not use single words but rather word co-occurrences as index terms. In a second step, the software artifacts are cluster based on the similarity of the corresponding documentation, and eventually the maintainer is provided with navigation means to browse the resulting categories. As a case study they categorize Unix tools based on their manual pages.

Merkl uses self organizing maps (SOM) to categorize MSDOS commands based on external documentation [Mer95]. Self organizing maps are an artificial intelligence (AI) technique based on neutral networks. SOMs are good at producing planar visualizations of high-dimensional data. A self organizing map is a planar grid of artificial neutrons, that able to reorganize its nodes based on an some strategy and feature vectors. Similar to the vector space model (VSM) this approach uses term frequencies as feature vectors and associates documents with points on the self organizing map.

Anquetil and Lethbridge rely on filenames as primary source of linguistic information [AL98]. They report that in the experiments conducted on their case study, a large legacy system written in the seventies, file names were a much better clustering criterion than function names or comments. The retrieval is based on a plain vector space model (VSM) approach with file names as document set. The authors use a combination of approaches to split up compound file names into terms, such as: external dictionaries, abbreviation detection strategies, and even looking at functions names in the source code.

Maletic and Marcus propose using Latent Semantic Indexing (LSI) to analyze software, which allows them to focus on source code as main source of linguistic information. In a first work they categorized the source files of the Mosaic web browser[MM00], and present in several follow-ups

other applications of Latent Semantic Indexing (LSI) in software analysis [MM01, MM03, MSRM04, MP05].

3.3 Traceability Recovery

In recent years information retrieval (IR) techniques haven been used to recover traceability links between documentation and software. Establishing links between natural language text documentation and source code is helpful in numerous tasks, such as navigating between documentation and source code in software comprehension and maintenance. In particular, traceability links between requirement specification and code are helpful to measure the completeness of an implementation with respect to stated requirements, and to infer requirement coverage of given source code.

Antoniol *et al* have published a series of papers on recovering code to documentation traceability [ACCL00, ACC+02]. They use two different information retrieval (IR) techniques, one based on a probabilistic model and one using a plain vector space model (VSM), which perform equally well. Both approaches rely on external documentation as text corpus. They argue that natural text has a richer vocabulary than source code and is thus more suitable as document set. The text corpus is then queried with identifiers taken from source code to get matching external documents.

Marcus and Maletic use Latent Semantic Indexing (LSI) to recover traceability links between source code and documentation [MM03]. They form the text corpus out of both documentation *and* source code, using identifier names and comments found in the source code as document content. In that way they do not query the text corpus with identifiers, but rather compute directly the similarity between external documents and source files. Each similarity higher than 0.7 is considered as a recovered link between source and documentation. They use the same case study as Antoniol *et al* and compare their results, which to some extent outperform the previous approach.

De Lucia *et al* introduce strategies to improve LSI-based traceability detection [LFOT04]. They use three techniques of link classification: taking the top-n search results, using a fix threshold of 0.75 or a variable threshold; of which the last performs best. Furthermore they create separate LSI spaces for different document categories and observe better results

that way, with best on pure natural language spaces and worst on pure source code spaces.

Cleland *et al* introduce strategies for improving the performance of a probabilistic traceability detection [CHSDZ05]. The retrieval is based on probabilistic network model, which is a directed acyclic graph with both documents and terms as nodes and term concurrencies as edges. It assumes a probability distribution of term concurrencies, and thus performs search queries by computing the probability that a query might appear in a document. On the one hand they present strategies that add additional edges to the model: They populate the graph with additional edges modeling hierarchy relationships among both document or term, or rather they enrich the model with edges between nodes which are known to belong to the same group or which are siblings. On the other hand, they remove edges based on user feedback. They apply the strategies on three case studies with varying performance gain.

Settimi *et al* find that recovering links from requirements to UML diagrams performs better than from requirements to source code [SCHK+04]. This due to the differences in the vocabulary: requirements and UML diagrams are both documentation and use a more abstract vocabulary than source code. Furthermore it is written from a different perspective, for example "the user views" compared to "the system displays", which can make it hard to recover traceability links.

Natt *et al*'s focus on external documentation only. They apply information retrieval (IR) techniques to detect duplication and interdependencies in requirements document [oDRC+02]. The retrieval is based on an improved vector space model (VSM), which exploits the specific structure of requirements documents. They experiment with different similarity measurements and thresholds, and plot for each precision and recall.

Di Lucca *et al* focus on external documents, doing automatic assignment of maintenance requests to teams [LPG02]. They compare approaches based on pattern matching and clustering to information retrieval (IR) techniques, of which clustering performs best. The information retrieval is done using a plain vector space model (VSM), which does not use any Latent Semantic Indexing (LSI).

Finally Huffman-Hayes *et al* compare the results of several information retrieval (IR) techniques in recovering links between document and source code to the results of a senior analyst [HHDO03]. The results suggest

that automatic recovery performs better than analysts, both in terms of precision and recall and with comparable signal-to-noise ratio.

3.4 Feature Location

Maintenance tasks are often centered around features, a usual maintenance task is about changing or adding functionalities related to the same feature. Therefore a basic and very helpful step in reverse engineering is to locate interesting features in the source code. Feature location can be formulated as identifying the relationship between the users view and the programmers view of the system. Most approaches addressing this problem rely on dynamic execution traces to locate the features.

Test-cases are commonly used to trace the features based on the assumption that each test-case covers one feature, whereas Zhao *et al* use information retrieval (IR) technology to detect the initial starting point of the feature traces [ZZL+04]. The retrieval is based on a plain vector space model (VSM) which does not use any Latent Semantic Indexing (LSI): external feature descriptions, given in natural language text, form the document set and *tf-idf* is applied as weighting scheme. The space is then queried with function signatures and the results arranged in a table and transposed such that we get ranked lists of functions. The authors apply this to avoid common functions being specific to all features.

Marcus and Maletic use Latent Semantic Indexing (LSI) to locate concepts in software, using LSI as a search engine and searching the source code for concepts formulated as search queries [MSRM04]. The retrieval is an interactive search process with relevance feedback from the user, and works on source code as text corpus. They compare the approach to regular expressions and program dependence graphs, and get better performance than with these approaches.

Even though Latent Semantic Indexing (LSI) is applicable at any level of granularity Marcus *et al* remained at the level of files and procedural systems. It is Kawaguchi *et al* that take it on a much larger scale. They categorize whole software projects in open-source software repositories [KGMI04]. For each project they group together all source files into one retrieval document, and use the identifier names as terms, skipping comments. Then they exploit LSIs capability to compare between terms

as well as documents: they cluster the terms and categorize the documents based on these clusters. The approach performs better, in terms of precision and recall, than leading approaches based on external documentation.

Furthermore they present MUDABlue, a tool that categorizes the projects and describes the categories with automatically retrieved labels. The tool uses cluster maps to visualize the categories and software projects [FSvH03].

3.5 Other Application of LSI

In [MM01] the authors use LSI to detect high-level conceptual clones, going beyond string based clone detection using LSIs capability to cope with synonymy and polysemy. They split the source files into procedures using these as retrieval documents. Then they cluster the procedures based on their similarity, and detect in a supervised process possible clones between files. Supervised meaning that their approach requires user intervention.

They argue that further automation would be to expensive and difficult, and thus human interaction is unavoidable. In their approach the user has first to select a set of documents as template, and then to examine the returned list of the possible clone candidates. Alas they do not support any user guidance to these process beyond lists with mere numbers: neither visualizations nor labels are used to guide the user.

Marcus *et al* use LSI to compute the cohesion of a class based on the semantic similarity of its methods [MP05]. In a case study they compare their semantic cohesion with several structure based cohesion measurements.

3.6 Software Clustering

A common way to reduce the complexity of a software system is to split it up into modules. But as a system evolves its structure decays due to architecture violations introduced by maintenance tasks [Par94, EGK+01].

Thus reengineering techniques are required to (re)modularize a system.

Clustering methods are a good starting point for the remodularization of software system, because the goal of clustering methods is to group related entities together. A general overview of clustering and its application in software reengineering is given in [Wig97] and a detailed overview of clustering algorithms themselves in [JMF99]. Another popular approach is Formal Concept Analysis (FCA).

Wu *et al* observe the stability of clustering approaches as a software system evolves [WHH05]. They apply six different approaches on subsequent versions of five open source systems, and log three criteria: stability, authoritativeness and the extremity of cluster distribution. The studied algorithms show distinct characteristics, for example the most stable is the least authoritative and vice versa. The authors conclude that current algorithms need significant improvement to support continuous monitoring.

Lindig *et al* modularize legacy systems based on the usage of global variables by procedures [LY97]. They use Formal Concept Analysis (FCA) to build a lattice with groups of procedures that use the same set of variables. They conduct three case studies, each written in a different procedural languages. The quality of the results correlate with the amount of modularization supported by the language of the system, hence the authors conclude with Wittgenstein's "Die Grenzen meiner Sprache sind die Grenzen meiner Welt".

Deursen *et al* investigate the usage of data records to port procedural systems to object orientation [vDK99]. Based on the observation that records in data structures are often not related to each other, they do not rely on the given structures but rather break them up and re-structure the records based on co-occurent usage. They use both a hierarchical clustering as well as Formal Concept Analysis (FCA), and provide in the conclusion a general discussion of dendrograms, which are the result of a clustering, and lattices, which are the result of FCA, in relation to software re-modularization.

Siff *et al* extend the previous approach including negative information as well [SR99]. They model both if a procedure uses a record as well as if a procedure does *not* use a record. Then they introduce the notion of concept partition, and present an algorithm to compute all possible partitions of the system given by the concept lattice. They prove the formal correctness

of the algorithm.

Tonella introduces a distance measurement between partitions to estimate the cost of restructuring the system [Ton01]. The distance is defined as the minimal number of elementary operation required to transform the first partition into the second partition. This is similar to the Mojo distance [TH99], but with another set of operations. The measurement is applied on twenty case studies using an approach based on Formal Concept Analysis (FCA) and a coupling-cohesion measurement. The authors verify some of the results by actually carrying out the proposed restructuring of the system. They conclude that such a refactoring is not a trivial task, but that the proposed partition has been a very useful suggestion.

Li *et al* try to exploit ownership by discriminating different coding styles [LY01]. Their assumption is that each author uses a distinct code formatting and dictinst naming conventions. An assumption which, of course, falls if a team adheres to a binding style convention. They sell their approach as a possibilty to split large projects into conceptual subsystems, making another assumption that each author is responsible for one module of the whole software only and explicitly.

Chapter 4

Semantic Clustering

"The second error is that of Lewis Carroll's Walrus who grouped shoes with ships and sealing wax, and cabbages with kings..."
— Reuben Abel

This chapter introduces *Semantic Clustering*, a novel 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.

Semantic clustering is a non-interactive and unsupervised technique. First Latent Semantic Indexing (LSI) is used to extract linguistic information from the source code and then clustering is applied to group related software artifacts into clusters. A cluster is a group of artifacts that use the same vocabulary. Therefore 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.

Finally, the inherently unnamed concepts are labeled with terms taken from the vocabulary of the source code. An automatic algorithm labels each cluster with is most related terms, and provides in this way 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,
- and *semantic links* between these clusters, which emphasize single software artifacts that interconnect the above domain concepts.

The following sections of this chapter explain *Semantic Clustering* step by step (see Figure 4.1). The "LAN-Simulation" [DVRG+05] is considered as an example to exemplify the explanations.

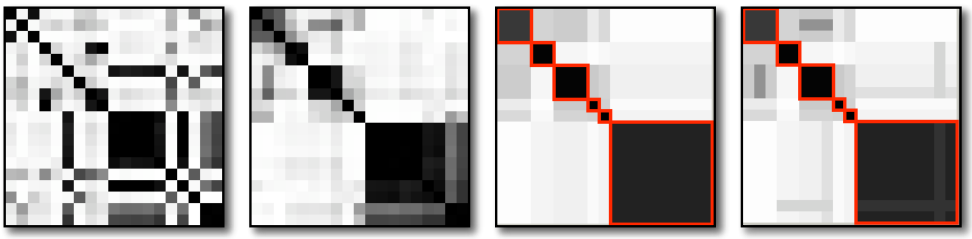

Figure 4.1: From left to right: unordered correlation matrix, then sorted by similarity, then grouped by clusters, and finally including semantic links.

4.1 Building the Text Corpus

Semantic clustering uses Latent Semantic Indexing (LSI) to build a semantic model of source code. LSI is an information retrieval (IR) technique that analyzes the distribution of terms over a text corpus (see Chapter 2). When applying LSI on a software system we break its source code into documents and we 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 [KGMI04].

We extract the vocabulary of source 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 use 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, or even none at all, more advanced algorithms and heuristics are required [CT99, AL98].

As proposed in Section 2.3 we exclude common stopwords from the vocabulary, as they do not help to discriminate documents, and use a stemming algorithm to reduce all words to their morphological root. An finally the term-document matrix is weighted with *tf-idf* (seeSection 2.3), to balance out the influence of very rare and very common terms.

4.2 Semantic Similarity

In the previous section we used Latent Semantic Indexing (LSI) to extract linguistic information from the source code. The result of this process is an LSI index $\mathcal{L}$ with similarities between software artifacts as well as terms. Based on the index we can determine the similarity between these elements.

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 (see Section 2.5) its values range between 0 and 1. The similarities between elements are arranged in a square matrix $\mathbf{A}$ called the *Correlation Matrix*.

To visualize the similarity values we map them to gray values: the darker, the 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. In short, the elements are arranged on the diagonal and the dots in the off-diagonal show the relationship between them—see the first matrix in Figure 4.1.

4.3 Semantic Clustering

Without proper ordering, the correlation matrix looks like 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 [Ber81]—therefore, we cluster the matrix such that similar elements are put near each other and dissimilar elements far apart of each other.

Furthermore applying a clustering algorithm groups similar elements together and aggregates them into concepts. Hence, a concept is characterized as a set of elements that use the same vocabulary.

Hierarchical clustering yields a tree, called *dendrogram*, that imposes both a sort order and a grouping on its leaf elements, see Figure 4.2 [JMF99].

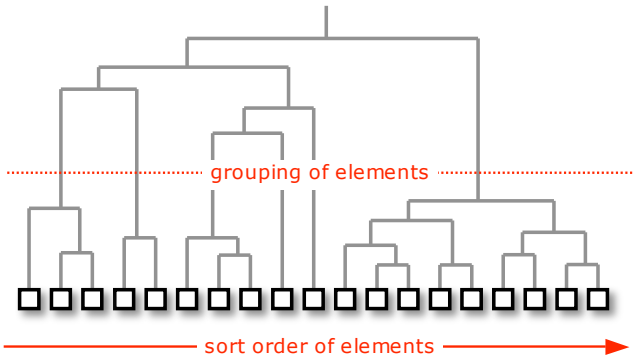

Figure 4.2: Hierarchical clustering yields a tree, called "dendrogram", that imposes both a sort order and a grouping on its leaf elements

Sort order. Traversing the dendrogram we collect its leaves and return them sorted by similarity, in that way we place similar elements near each other and dissimilar elements far apart of each other. 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 in this paper are ordered using *average linkage* clustering, and traversing the tree *large-clusters-first*.

Clustering. To actually break down the system into a hard partition of clusters, the dendrogram tree is cut off at a given threshold and all remaining leaves taken as clusters, as illustrated by the dotted line on Figure 4.2. Since both sort order and grouping are taken from the same dendrogram, it is guaranteed that the elements of each clusters are in a row.

Using these two results we reorder the matrix, group the dots by clusters and color them with their average cluster similarity—as illustrated on the

second and third matrices on Figure 4.1.

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 $(|R_i|, |R_j|)$. In short, 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 4.1.

4.4 Semantic Links

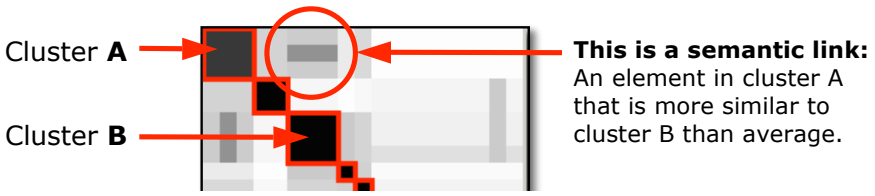

Figure 4.3: A *semantic link* is a one-to-many relation: A document in cluster A that is more similar to those in cluster B than all its siblings in A.

The tradeoff of the clustering is, as with any abstraction, that some valuable detail information is lost. We use *semantic linking* to pick out relations at the level of elements, and plot them on top of a clustered correlation matrix.

Our experiments showed that 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, we plot d_n on top of the clustered matrix: 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 4.3 gives an example of a semantic link.

The case-study in Section 6.1 uses a variation of this technique to reveal high-level conceptual clones [MM01] instead of "semantic links". The setup of the correlation matrix in that case-study is basically the same as presented in this chapter, with the difference that the elements are first grouped by top-level modules and then clustered within these modules only.

4.5 Correlation Matrix

This section summarizes the previous steps: We use a shaded correlation matrix to illustrate the semantic clustering of a system. A correlation matrix is gray-scale raster-graphic: each dot $a_{i,j}$ shows the similarity between element[1] d_i and element d_j—the darker, the more similar. In other words, the elements are arranged on the diagonal while the dots in the off-diagonal show the relationship between them, see Figure 4.4.

Alas, 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 colored with their average similarity, that is the semantic cohesion [MP05]. This offers a high-level view on that system, abstracting from elements to concepts.

A sample *Correlation Matrix* is shown in Figure 4.4. There are three annotated clusters: A, B and C. On the visualization we can see:

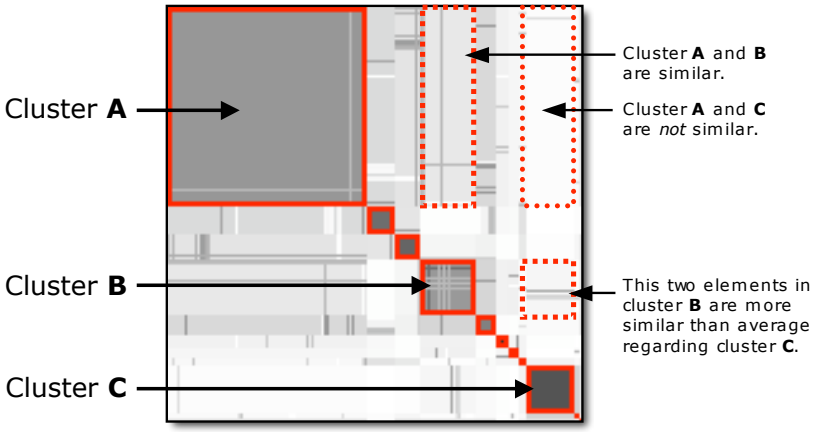

Figure 4.4: A sample *Correlation Matrix*: Cluster A and B are similar, cluster C is very cohesive, two elements in B are related to C.

Semantic Similarity. The colors 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.

[1] In the scope of this chapter the term *element* refers to software artifacts, such as for example classes, methods or functions. But due to the interchangeability of documents and terms in the vector space model (VSM), see Section 2.2, anything said about elements holds true for both documents *and* terms.

Semantic Cohesion. The color 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. On Figure 4.4, 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, that is how the semantic concepts are related to each other. The relation between the semantics and the structure on the other hand is discussed and visualized in Chapter 5.

4.6 Labeling the Clusters

Just visualizing clusters is not enough, we want to have an interpretation of their of semantic concepts. We need a written description of the concept covered by a cluster, that is we need labels that describe the cluster. Often just enumerating the names of the software artifacts in a cluster (for example displaying the class names) gives a sufficient interpretation. But, if the names are badly chosen or in case of analyzing unnamed software artifacts, we need an automatic way to identify labels. Figure 4.5 shows the labels for the concepts found in the LAN example.

The labeling works as follows: As we already have an LSI-index at hand, we use this index as a search engine [BDO95]. We reverse the usual search process where a search query of terms is used to find documents, and instead, we use the documents in a cluster as search query to find the most similar terms. To obtain the most relevant labels we compare the similar terms of the current cluster with the similar terms of all other clusters.

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 $A \in \mathcal{A}$. Given the similarity between a term t and a cluster A as $\text{sim}(t, A)$, we define the relevance of

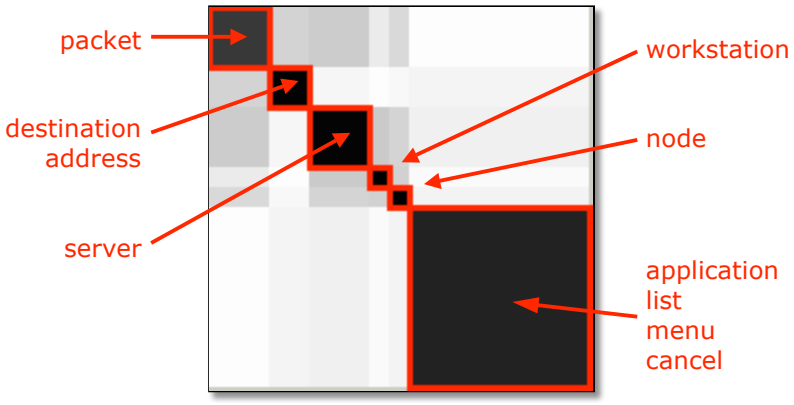

Figure 4.5: Automatically retrieved labels describe the concepts. The labels were retrieved using the documents in a concept cluster as query to search the LSI space for related terms.

term t_0 according to cluster A_0 as follows:

$$\mathrm{rel}(t_0, A_0) = \mathrm{sim}(t_0, A_0) - \frac{1}{|\mathcal{A}|} \sum_{A_n \in \mathcal{A}} \mathrm{sim}(t_0, A_n)$$

This raises better results than just retrieving the top most similar terms [KDG05]. It emphasizes terms that are specific to the current cluster over common terms.

Chapter 5

Distribution of Concepts

"A man without a name is but a shell.
It is the name to which his soul is tied."
— a shamanist saying

The semantic clusters help us grasp the concepts implemented in the source code. However, the clustering does not take the structure of the system into account. As such, an important question is: How are these concepts distributed over the system?

To answer this question, we use the metaphor of a distribution map [Ber73]. In this chapter we discuss how the semantic concepts are related to the systems structure. 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 we find both, concepts that correspond to the structure as well as concepts that cross-cut it.

5.1 The Distribution Map

To illustrate the correlation between the semantic clustering and the structural modularization of a system we introduce the *Distribution Map*[1], see Figure 5.1. It is composed of large rectangles containing small squares in different colors. For each structural module there is a large rectangle and within those for each software artifact a small square. The color of the squares refers to the semantic concepts implements by these artifacts.

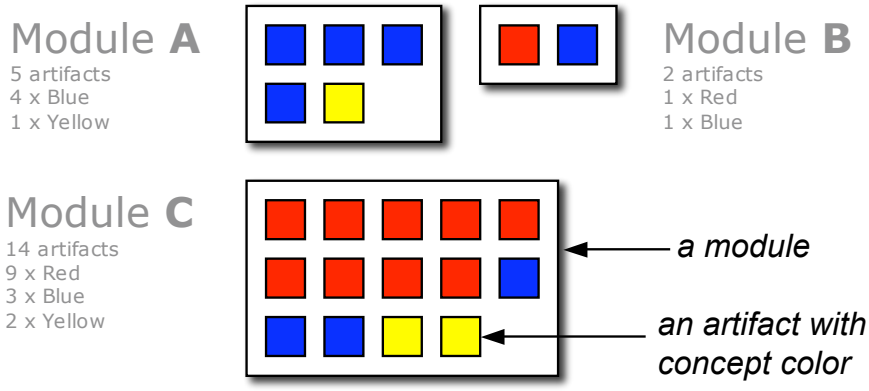

Figure 5.1: The *Distribution Map* shows how semantic concepts are distributed over the structural modularization of a system. Boxes denote modules and software artifacts, colors refer to concepts.

The choice of colors is crucial to the readability of the *Distribution Map*. We need to pay attention to the characteristic of human vision to make sure that the visualization easy to read and to avoid wrong conclusions [**?**]. First, scattered squares are easier to spot if they are shown in distinct colors than if they are shown in colors similar to the well-encapsulated colors. Therefore light colors are a good choice for cross-cutting concerns and dark colors a good choice for well encapsulated concepts. Second, a human reader will draw conclusions concerning the similarity between concepts based on the similarity of the colors 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 colors, such that more similar concepts use more similar colors and vice versa.

[1]This visualization is by no means restricted to semantics and structure, it is generally applicable on any set of entities with two different partitions [DGK06].

5.2 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 we find both, concepts that correspond to the structure as well as concepts that cross-cut it. In this section we propose a vocabulary to describe the most common distribution patterns (see Figure 5.2):

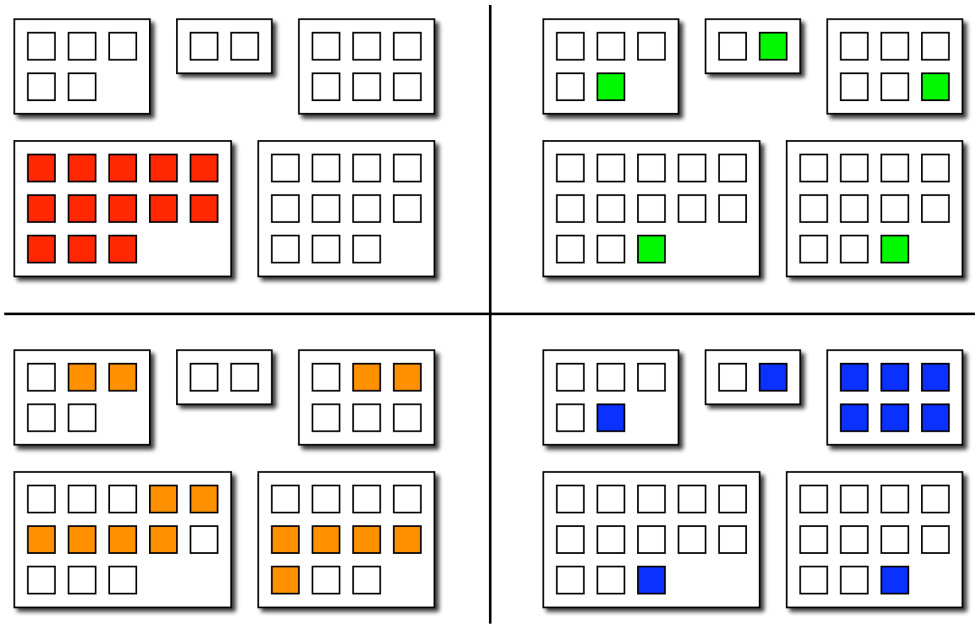

Figure 5.2: From top left to bottom right: a well-encapsulated concept that corresponds to the modularization, a cross-cutting concept, a design smell scattered across the system without much purpose, and finally an octopus concept which touches all modules.

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 5.2 the red concept illustrates this pattern, but not the orange one.

Cross-Cutting Concepts – if a concept cross-cuts the structure, we call this a *cross-cutting concept*. Such a concept spreads across multiple

modules, but includes only one or very few artifacts within each module. In Figure 5.2 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 5.2 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.

5.3 Case-Study

This section discusses the concept distribution considering a three-tiered application as example. *Outsight* is a web based business application written in Java and about 200 classes large. It is a web-based job market, a search engine where graduate students enter their CV and business companies submit profiles. The package structure splits the system into three layers and some separate modules. The layers are named *database*, *logic* and *presentation*, furthermore there are three *migration* and two *utility* packages.

We analysed the system at class level, used an *tf-idf* weighting, applied LSI with a rank of $k = 12$ and clustered the documents using a threshold of

$\vartheta = 0.5$, which resulted in nine distinct concept clusters. Figure 5.4 lists all the concepts and their labels, and Figure 5.3 shows the *Distribution Map* of the system.

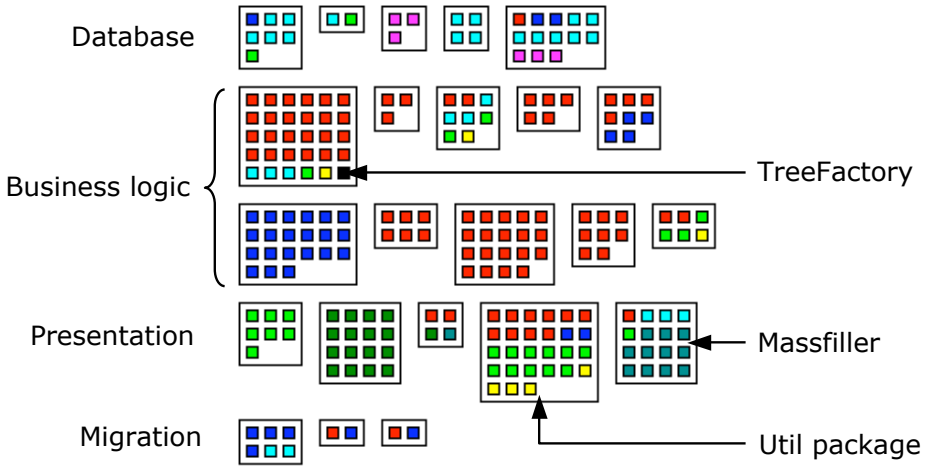

Figure 5.3: This *Distribution Map* illustrates the distribution of semantic concepts over packages and classes of the Outsight case-study. There are both encapsulated and cross-cutting concepts.

Concept	Main Module	Top 7 Labels
Red	businesslogic	logic node ID user state admin login
Blue	matching	language match profile create content store candidate
Cyan	database	data set create fail database row equal define
Darkcyan	massfiller	company mass region math store random perform
Black	TreeFactory	logic node child remove trigger tree set children
Green	util	filename number size file stream system param
Yellow	util	time run equal date number end main
Magenta	data	debug driver delay pool connection runtime SQL
Darkgreen	candidate	document written presentation add output candidate PDF

Figure 5.4: All semantic concepts of the Outsight case-study: the *top seven labels* column lists the most relevant terms of each concept, and *main module* column lists the package that contains most of its classes.

Self-contained Concepts – the system has two main concepts: Red and Blue are the largest concepts of the application. Together they make up 80% of the *business logic* layer, clearly predominating not only this part but the whole system. Both show good encapsulation: Blue coincides with one package and Red with four.

Noteworthy are the appearances of Red and Blue in the *migration* packages. In this case, linguistic information reveals details the evolution of the system: each of the packages contains scripts to move

from one version of the system to the next. For example, because we find concepts Red and Blue in the package of the first migration, we can conclude that these concepts changed significantly from the first to the second version of the system.

Concept **Red** implements the main structure of the business logic, and its labels and vocabulary indicate that the applications uses a generic tree based data structure. It has an alarming number of outliers in the *utility* package, a manual check has to show if this is a structural flaw or not. Maybe it is not, as it is inevitable that some utility classes, which are almost exclusively used by the main concept, end up in the main concept even if a human expert would categorize them apart. But in this case the *utility* package is divided into equals parts of Red and Green, which indicates that it should be split into two packages.

Concept **Blue** implements the search engine that matches profiles against candidates. There are three outliers on the *database* layer, which shows that this concept is more deeply involved on that low level layer than the more general Red concept. Moreover, there are no outliers of Blue on the *presentation* layer one level above.

Solitary Concepts – Beside the main concepts the system has two other well-encapsulated concepts. Darkgreen and Darkcyan are almost perfect instances of a solitary concept: both are well encapsulated by one package.

Concept **Darkgreen** is a part of the presentation layer and generates, as its labels convey, PDF documents instead of web pages. Its package should be renamed, as *candidate* is a misleading name for a PDF engine.

Concept **Darkcyan** is called *massfiller* and is involved on the database layer, as the three Cyan outliers indicate. It is likely to be a tool to fill the database with random data.

Cross-Cutting Concept – the system has one clear cross-cutting concept, which is Yellow.

Concept **Yellow** is a meta concept. Its first label is *time*, standing out of the list with a similarity of $\delta_1 = 0.92$ compared to the remaining labels with $\delta_2 = 0.83$ and below. In fact all its classes deal with

time handling, but each class provides distinct functionality specific to its package. Moving these classes into their own package therefore does not make much sense.

Octopus Concepts – the system has two octopus concepts: concept Cyan and Green, which are both badly designed. The base packages of Cyan should be merged, the base package of Green should be split.

Concept **Cyan** is, as the labels reveal, the *database* concept. Most of its classes are on that layer, yet they are spread across several package: this layer should be cleaned up. But otherwise it is a well designed octopus concept: it is used on the *business logic* layer and by the *massfiller* solitary, but not on the *presentation* layer.

Concept **Green** is, as the labels reveal, the *file* and *IO* concept. It is mainly used on the *presentation* layer, but has outliers on all layers. It shares its base package with concept Red, which indicates that this package should be split into two.

Design Smells – some design smells have already been addressed, remaining are two packages: Magenta and Black. The first is a badly encapsulated solitary concept, and the latter a "Black Sheep".

Concept **Magenta** provides, as the labels convey, very low level database functionality. Its classes are distributed over two packages, and should be merged into one package to make it onto a solitary concept.

Concept **Black** is restricted to one single class, this smells. Its structure and its evolution confirm this assumption: it is not only the class with most lines of code but also the class with most CVS revisions. In fact the class is named *TreeFactory* and is used to create all data objects of the business logic layer. Every time the business model or process changes this class has to change also.

To summarize our findings: We showed how to use the *Distribution Map* to gain insight into the main concepts of the system. Furthermore the linguistic analysis pointed out several design flaws, and proposed refactorings to solve them: rearrange the modularization of the database layer, split the utility package into two halves and tackle the TreeFactory class to find a better solution for object creation.

Chapter 6

Experiments

"Stat rosa pristina nomine, nomina nuda tenemus.[1]*"*
— Umberto Eco, in *The Name of the Rose*

To show the generic nature of the approach we apply it at different levels of abstraction and on case studies of different sizes and written in two different languages:

1. In the first case-study we analyze the core and the plug-ins of a framework, the Moose reengineering environment [DGLD05]. This experiment focuses on the relation between architecture and semantics. It reveals four cases of duplicated code and a core functionality misplaced in one of the plug-ins.

2. The second case-study is the class MSEModel, which is one of the largest classes in Moose. In this experiment we analyze the relationship among the methods of MSEModel. The experiment reveals that the class should be split as it serves at least two different purposes.

3. In the third case-study, the JBoss open-source Java application server, we focus on the distribution of concepts over classes and show the

[1] The ancient rose continues to exist through its name, yet its name is all that remains to us.

strength of our approach in identifying and labeling domain concepts.

4. In the fourth case-study we analyze the distribution of the semantic concepts over the package structure of Azureus, a popular file sharing client. The experiment presents a software system with very well-encapsulated concepts.

The following table summarizes the problem size of each case study. It lists the number of documents and terms in the vector-space-model, and the rank to which the vector space has been broken down with LSI (see Chapter 2). In the case of Moose, JEdit and Azureus we use classes as input documents, while in MSEModel we use methods.

Case-study	Language	Type	Documents	Terms	LSI-Rank	Threshold
Moose	Smalltalk	*Classes*	726	11785	27	N/A
MSEModel[2]	Smalltalk	*Methods*	4324	2600	32	0.75
JBoss	Java	*Classes*	660	1379	16	0.5
Azureus	Java	*Classes*	2184	1980	22	0.4
JEdit	Java	*Classes*	394	1603	15	0.5
Ant	Java	*Classes*	665	1787	17	0.4

Figure 6.1: The case studies and their characteristics.

6.1 Semantic Links between a Framework and its Plug-ins

This case-study shows the application of our approach to analyze how modules are semantically related to each other. The granularity of the correlation matrix are classes, grouped by modules and ordered inside modules by semantic similarity. The goal here is to detect relationships between the plug-ins and the framework. One would expect to find for each plug-in a large part of classes that are not similar to the framework or other plug-ins, since each plug-in extends the framework with new functionality. One would also expect to find some classes that share semantic content with the framework, since each plug-in hooks into the core.

Figure 6.2 shows the correlation matrix. There are five blocks on the main diagonal, one for each module. They are, from top-left to bottom-right: Hapax, Van, Moose, ConAn and CodeCrawler. Moose is the core

[2]This case-study focuses on 164 out of its 4342 methods, for example those methods that belong to the *MSEModel* class (see Section 6.2).

framework, all other module are plug-ins built on top of Moose [DGLD05]; CodeCrawler extends Moose with visualization capabilities [LD05].

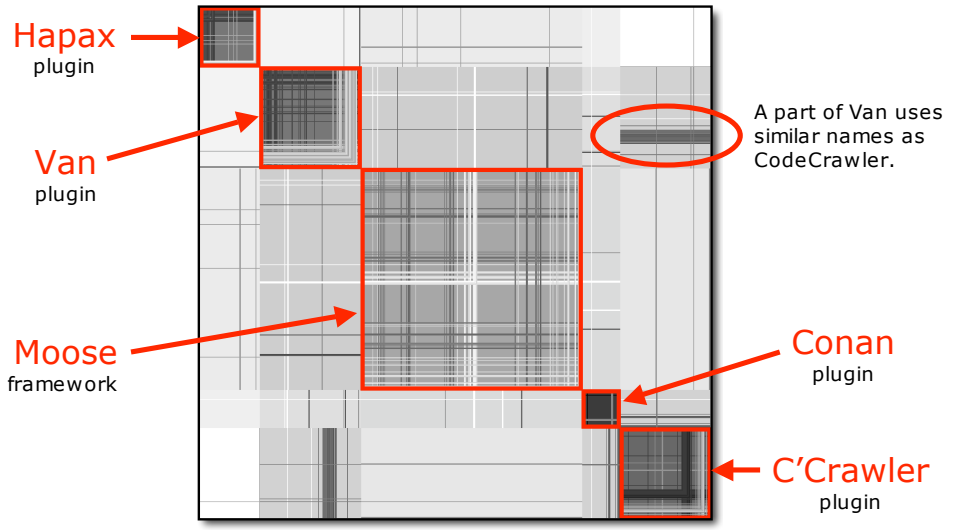

Figure 6.2: The correlation matrix of the Moose environment and four plug-ins.

The background color of a cluster shows its density given by the average similarity between all its classes. The background colors of the off-diagonal rectangles show the relationships between clusters (that is the average similarity between all classes of one cluster to all classes of the other cluster). If the background color is white two clusters are not related, if it is gray they are related – the darker the gray the stronger the relationship.

The lines indicate *semantic links* which are single classes in one cluster that stand out as its similarity to all classes in another cluster is significantly above or below the average similarity.

Hapax. The background color shows that Hapax is slightly related to the core, but not related to the other plug-ins. A noteworthy semantic link is the line parallel to the core which indicates a class in Hapax strongly related to the core as a whole. Another semantic link is shown by the line orthogonal to Van indicates a class in Van strongly related to Hapax. Closer inspection reveals that:

- the first is a generic *visitor* class missing in the core that got implemented in Hapax, and

- the second is an implementation of a *matrix* data structure in Van duplicating code in Hapax.

Although Hapax implements several visualizations (for example the one in this article) it does not share any semantics with CodeCrawler which is a generic visualization tool. This is an indicator that its visualizations are not based on CodeCrawler, and that it could become a source of potential functional duplication.

Van. The broad white margin, at the bottom right inside its box, is an indicator that Van contains many *utility* classes not related to the main concepts implemented in it. The background colors of the off-diagonal rectangles show that Van is related to both the Moose core and CodeCrawler. Noteworthy semantic links are – beside the line shared with Hapax mentioned above – the fat group of lines parallel to CodeCrawler; and the lines parallel to ConAn, since Van is otherwise not related to ConAn. Closer inspection reveals that:

- the first as *subclasses* extending the CodeCrawler framework, thus Van makes heavy use of that plug-in, and
- the second link is an implementation of a *tree* data structure duplicating code in ConAn.

Moose. As we discuss the relationship of Moose to its plug-ins in the other paragraphs, we examine here its inner structure. The light background color shows that Moose is less semantically cohesive than the plug-ins. While all plug-ins deal with one concept, ConAn being the most cohesive, Moose contains two broad concepts separated by the white lines in its center. Closer inspection reveals that the first concept is the meta-model, including the FAMIX model [DTD01]; and the second concept consists of metrics and operators common to these metrics.

ConAn. The background colors show that ConAn is, as Van, related to both the core and CodeCrawler. Noteworthy semantic links are – beside the code duplication mentioned above at Van – the group of lines parallel to CodeCrawler; and the white cross inside the box, standing out from the otherwise very cohesive content. Closer inspection reveals:

- the first, again as in the case of Van, as *extensions by subclassing* of the CodeCrawler framework, and

- the second as *user-interface* classes.

CodeCrawler. The background colors show that CodeCrawler is more related to other plug-ins than to the core, thus revealing it as being more an extension of the core than a real plug-in. Which is the case, since CodeCrawler extends Moose with visualization capabilities. Some noteworthy semantic links have already been mention above, the two remaining are the long line parallel to the the core stretching over all other plug-ins; and the dark cross inside its box. Closer inspection reveals that the first as *CCItemPlugin* the root class of the hierarchy extended by both Van and ConAn, and the second as 2D-geometry classes, forming the core of CodeCrawler *rendering engine*.

Our findings in this case study show that the correlation matrix reveals valuable information about the relationship between the different modules: it has revealed about a dozen strong relations by inspecting the semantic link. Among them, we found four cases of potential code duplication [MM01], and one case of functionality missing in the core.

6.2 Semantic Clusters of Methods in a Class

In this experiment we apply our approach to understand the internals of a single class (see Figure 6.3). The level of abstraction are methods, that is the documents of the LSI-space are method bodies. The goal is to detect different concepts in a large class. The class is *MSEModel*, with 164 methods. It the core of Moose, from the previous case study, and its purpose is similar to the *Document* class in the XML DOM model [WSB+98].

The most evident patterns are: the large cluster in the top left, and the small but cohesive cluster in the bottom right. As the background colors show all other clusters are related to the large one, but none of them to the cohesive one. A look at the method names reveals the large cluster are the *accessors* of the class and provides access to the nodes in the model's graph. The cohesive cluster is about *meta-information*, it provides access to information, such as, the creation date of the model or the version of the meta-model.

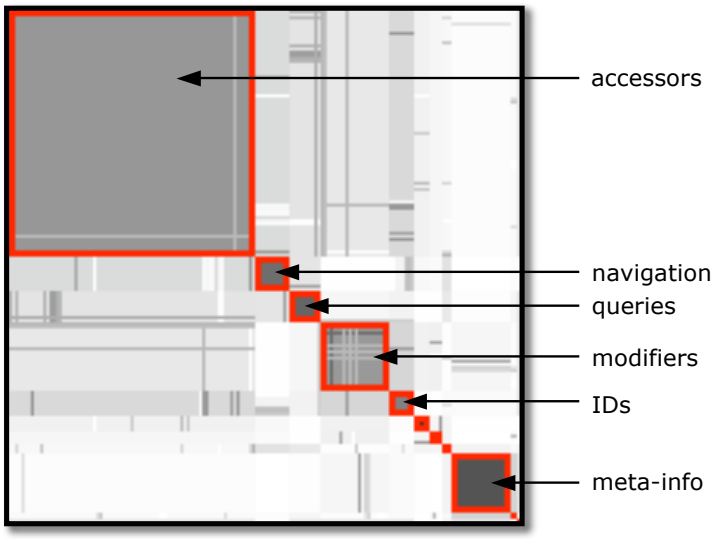

Figure 6.3: The correlation matrix of the methods inside the MSEModel class.

We discuss the remaining clusters in their order on the diagonal from top-left to bottom-right: The second and the third cluster are quite small and deal with structural relationships, the labels of the latter are for example: *inheritance, subclass, superclass, hierarchy,* and *override*. The fourth cluster is medium-sized are *modifiers*, containing methods to manipulate the model (for example *addEntity* and *removeEntity*). As the off-diagonal shows, the modifiers are related to the accessors but not to structural relationships. The fifth cluster however is related to the modifiers, and its labels reveal that it is dealing with unique IDs of model nodes. Its top labels are: *uuid, MSEUUID, id*.

Next are three clusters not related to any other, their top labels are *log, stream, cr* and *import, context, facade* and *space, term, vocabulary*. Evidentally, they are the *logging* facility, the *import* facility, and an extension of the Moose model specific to LSI and semantic clustering.

In this case study, we looked at the clusters and their labels and got a full overview of the concepts covered by this class. Furthermore, we identified four concepts not related to the core functionality of the class, that might be good candidates to factor out into classes of their own.

6.3 Distribution of Semantic Clusters in JBoss

This experiment focuses on the distribution of concepts over classes. As a case study we use JBoss, an open-source Java application server. We applied semantic clustering with a threshold of $\delta = 0.5$, which yields ten distinct concepts that are illustrated on Figure 6.4. The automatically retrieved labels are listed on Figure 6.5.

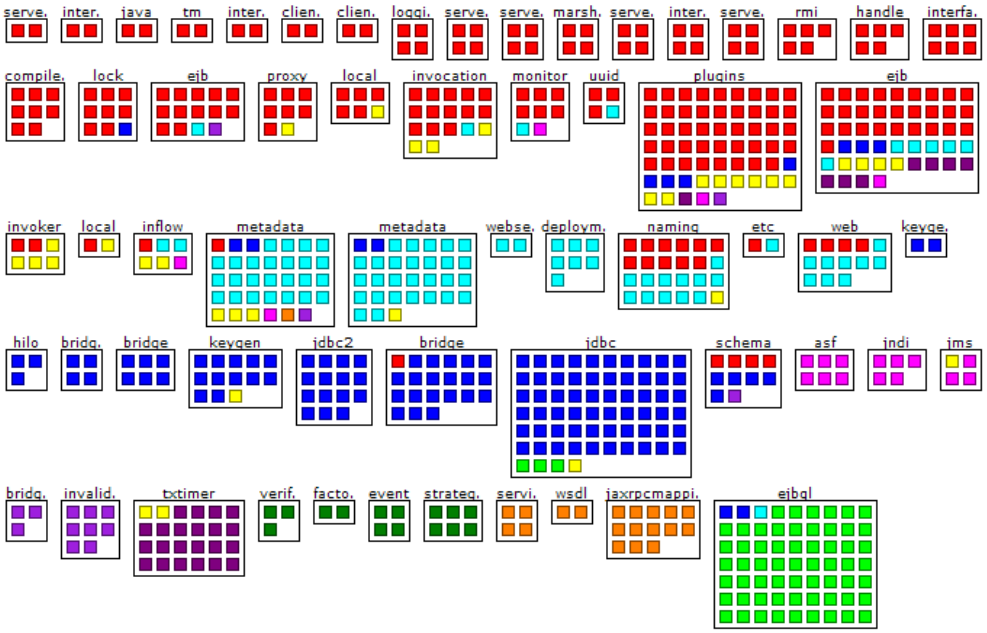

Figure 6.4: The distribution map of the semantic clusters found in JBoss.

Color	Size	Labels
red	223	invocation, invoke, wire, interceptor, call, chain, proxy, share
blue	141	jdbccmp, JDBC, cmp, field, loubyansky, table, fetch
cyan	97	app, web, deploy, undeployed, enc, JAR, servlet
green	63	datetime, parenthesis, arithmetic, negative, mult, div, AST
yellow	35	security, authenticate, subject, realm, made, principle, sec
dark magenta	30	expire, apr, timer, txtimer, duration, recreation, elapsed
magenta	20	ASF, alive, topic, mq, dlq, consume, letter
orange	20	qname, anonymous, jaxrpcmap, aux, xb, xmln, WSDL
purple	16	invalid, cost, September, subscribe, emitt, asynchron, IG
dark green	15	verify, license, warranty, foundation, USA, lesser, fit

Figure 6.5: The labels of the semantic clusters of JBoss.

While Concept Red, which is the largest cluster and thus the main concept

of the system, is labeled with terms such as *invocation, interceptor, proxy* and *share*, Concept Cyan covers the deployment facility.

The most well encapsulated concepts are Dark green, Orange, Green and also Blue. The first three are placed apart from Red, which is the main concept of the system, while Blue has outliers in the red core packages. From the labels and package names we conclude that Dark green is a bean verifier, that Orange implements JAX-RPC and WDSL (for example web-services) and that Green as an SQL parser and that Blue provides JDBC (for example database access) support.

The most cross-cutting concept is Yellow and spreads across half of the system. The labels leave no doubt that this is the security aspect of JBoss.

Noteworthy is the label *loubyansky*. Evidentally, this is the proper name of one of the developers and as his names pops up among the labels of concept Blue, we can assume that he is one of the main developers of that part of the system. And in fact, further investigation proves this true.

Noteworthy as well are the labels of concept Dark green, as they expose a failure in the preprocessing of the input data. To exclude copyright disclaimers, as for example the GPL licence, we ignore any comment above the the *package* statement of a Java class. However, in the case of the three packages covered by concept Dark green this heuristic failed, as they contain another licence within the body of the class.

In this case study, we analyzed how semantic clusters distribute over the package structure of JBoss. We identified some domain concepts, most of them well known EJB technologies, and security as a cross-cutting aspect. Additionally we identified one of the system's main authors, and spotted a failure in the preprocessing of the input data.

6.4 Distribution of Semantic Clusters in Azureus

In this experiment we show how well the distribution map scales up. As a case study we use Azureus, a popular bittorent client, that has about

2000 classes. We applied semantic clustering with a threshold of $\delta = 0.4$, which yields 14 semantic clusters that are visualized on Figure 6.6.

Figure 6.6: The distribution map of the semantic clusters found in Azureus.

Figure 6.6 is noteworthy for three reasons:

- The distribution map shows an extraordinary match between the concepts and the package structure. There are no cross-cutting concepts. Further investigation is necessary to decide, whether this is due to good design or due to strict code ownership imposing author specific vocabularies in the system, or maybe, even both explanations

apply.

- The *util* package (for example the third box in the third row) assembles classes from a variety of concepts. We call such a package a *concept assembler*, utility packages such as this one often follow this pattern.

- Concept Red is a good example of an *octopus concept*. It the largest cluster of the system and its "tentacles" reach into other concepts such as for example Blue. What we observe here is the layering of the system: concept Red, as the largest cluster of the system, implements the system's main domain concept; while cluster Blue is, as packages names and labels indicate, implements the user interface.

In this case study, we analyzed how semantic clusters distribute over the package structure of Azureus. We shows that the distribution map scales up to thousands of elements, presented three characteristic patterns and showed how semantic clustering is able to reveal architectural layers.

6.5 Distribution of Semantic Clusters in JEdit

JEdit[3] 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, applying semantic clustering resulted in nine domain concepts. Figure 6.7 illustrates how the retrieved domain concepts are distributed over the package structure: the parts are the packages, the elements are the classes and the colors refer to the detected clusters.

The table below lists for each concept its size, its spread, its focus and a description of its concept. The concepts description were created using was the following process:

- For well-encapsulated concepts, as for example Blue, the description has been derived from the package name.

- In the case of cross-cutting concepts, as for example Yellow or Pink,

[3]JEdit version 4.2, http://www.jedit.org/.

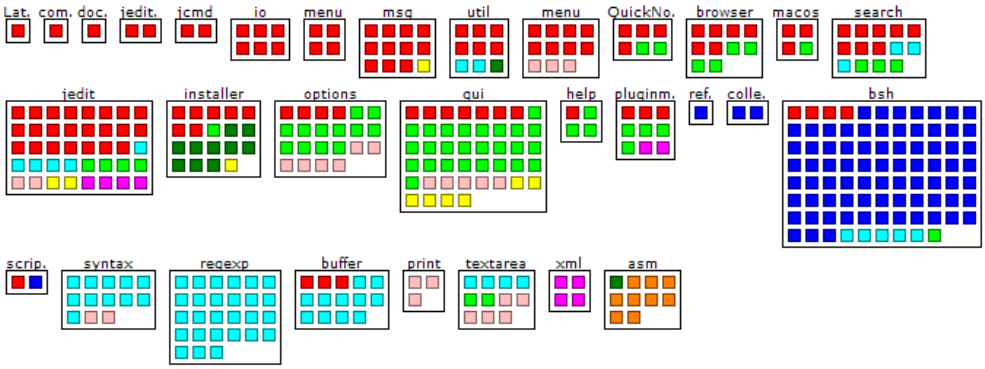

Figure 6.7: Distribution Map of linguistic concepts over the packages of JEdit 31 parts, 394 elements and 9 properties).

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.

color	**size**	**concept**
red	116	(main domain concept)
blue	80	BeanShell scripting
cyan	68	regular experssions
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

For example 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.

On Figure 6.7 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 than for example concept Cyan, which implements regular expressions.

The three most 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:

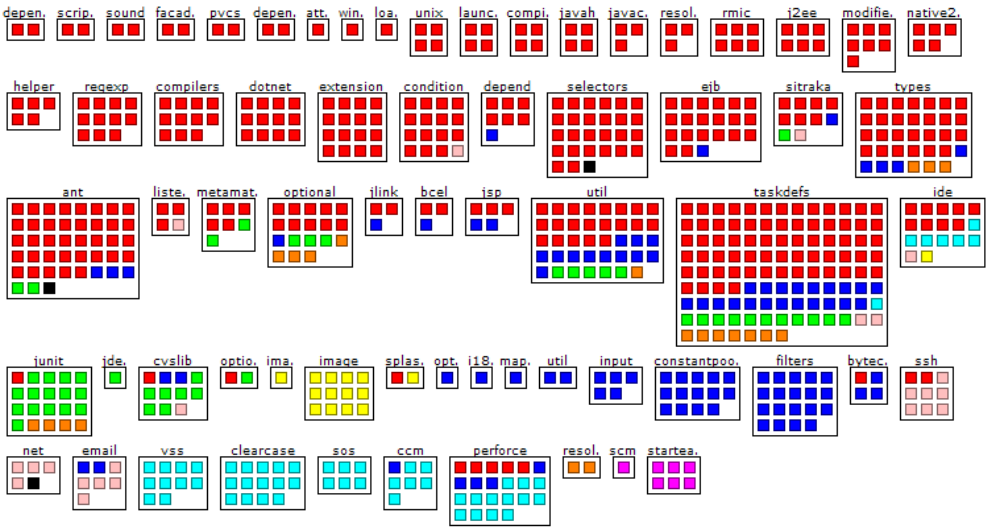

Figure 6.8: Distribution Map of linguistic concepts over the packages of Ant (66 parts, 665 elements and 9 properties).

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 refactor them into packages of their own.

6.6 Distribution of Semantic Clusters in Ant

Ant[4] is a popular development tool in Java. Its source has a total of 665 classes in 66 packages and uses a vocabulary of 1787 distinct terms. Applying semantic clustering resulted in nine domain concepts. Figure 6.8 illustrates how the retrieved domain concepts are distributed over the given package structure.

[4]Ant version 1.6.5, http://ant.apache.org/.

color	size	concept
red	390	(main domain concept)
blue	103	string processing
cyan	56	version control clients
green	48	unit-test support
pink	23	network protocols
orange	21	XML handling
yellow	15	image and graphics
magenta	7	another versioning client
black	3	FTP and filesystem

The table above lists for each concept its size, its spread, its focus and a description of its concept. The concept description was obtained using a process similar to the one describe in Section 6.5. For example the labels for the Orange cluster are: *entity, SAX, parser, jaxp, XML, factory, feature, systemid, catalog* and *XSL*—thus we assigned the concept "XML handling" to this cluster.

The three concepts with the lowest encapsulation (Blue, Green and Orange,)broadly cross-cut the package structure and implement main features of a development tool: string processing, unit-testing and XML handling. Other features such as, access to version control systems and network protocols, are well-encapsulated in concepts.

The order of the packages is not arbitrary or by name, but reflects the distribution of the concepts. On the first three rows there are packages that implement the core functionality of Ant, the largest concept which is Red, dominates this part of the figure. Then, on row four we find the octopus concepts Blue and Green that are used by the core packages in the third row. And, on the last row there are well-encapsulated plug-ins such as the Pink parts, which implement network protocols, or the Cyan and Magenta parts, which implement plug-ins for different version control systems.

Chapter 7

Discussion

"Gefühl ist alles, Name ist Schall und Rauch"
— Johann Wolfgang von Goethe,
Faust's answer to the "Gretchenfrage'

In this chapter we discuss the different variations of the approach.

On the granularity of software artifacts. The approach takes text documents as input. The source code can be broken into documents at any level of abstraction. Straightforward approaches are breaking it into classes or methods, like presented in this paper. However, other slicing solutions are possible as for example execution traces [KGG05].

On the parameters used. Our approach depends on several parameters, which are hard too choose for someone not familiar with the underlying technologies. First, LSI depends on the weighting functions and the choice of the rank. We weight the term-document-matrix with *tf-idf*, to balance out the influence of very rare and very common terms [Dum91]. For the choice of the rank, usually observed good results with this heuristic: $r = (m * n)^{0.2}$.

The clustering is parameterized as well. The clustering uses a hierarchical *average-linkage* clustering and takes either similarity threshold or a fix number of clusters as parameter, using a threshold between $\delta = 0.75$ and

$\delta = 0.5$ yields good. And finally a semantics link is an element in cluster d_1 with a different similarity to cluster d_2 than average, typically we use a threshold of $\delta = 0.2$ to decide this.

On badly named identifiers. Not unlike structural analysis which depends on correct syntax, semantic analysis is sensitive to the quality of identifiers and comments. Software systems with a good naming convention and well chosen identifiers yield best results. In case of legacy code that uses other naming conventions, or even none at all, please refer to [CT99] for suitable algorithms and heuristics.

Our approach recovers the developer knowledge put into the identifiers and comments, but if the developers did not name the identifiers with care, our approach fails since valuable developer knowledge is missing. For example if variables are just named *temp*, *foo* or x, y and z. Due to the strength of LSI in detecting synonymy and polysemy[1], our approach can deal with a certain amount of such ambiguous or even complectly wrong named identifiers – but if most of the identifiers in the system are badly chosen, the approach fails.

On the abbreviated identifiers. This is similar to badly named identifiers. Yet, LSI analyzes the statistical distribution of terms across the documents, it is therefore for the clustering not relevant whether identifiers are consistently written out or abbreviated. But if the labeling task comes up with terms like for example *pma*, *tcm*, *IPFWDIF* or *sccpsn* this does not tell a human reader much about the system[2]. Please refer to [AL98] for approaches on how to recover abbreviations.

On the dimension of the vocabulary. The vocabulary of source code is very small, smaller than that of a natural language text corpus. Intuitively explained: LSI mimics the way children acquire language [LD91], and a human with a vocabulary of 2000 terms is less eloquent and knowledgeable than a human with a vocabulary of 20'000 term. The smaller the vocabulary, the stronger the effect of missing or incorrect terms.

In average there are only about 5-10 distinct terms per method body, and 20-50 distinct terms per class. In a well commented software system, these

[1] Synonymy refers to different words having the same meaning and polysemy refers to the same word having different meanings.

[2] These terms are examples taken from a real case study not included in this paper, where about a third of all identifiers where abbreviations. In this case the labeling was completely useless.

numbers are higher since comments are human-readable text. Thus LSI does not perform as perfectly on source code as on natural language text [LFOT04], however the results are of sufficient quality.

On the use of ontology. As LSI is not based on an ontological database, its vocabulary is limited to the terms found in source code. In case of missing terms, our approach will not find accurate labels. Take for example a text editor in whose source code the term *text-editor* is never actually used, but terms like *file* and *user*. In this case our approach will label the package with these terms, as a more generic term is missing. Thus, using an ontology might improve our results.

On using the approach to gain a first impression. When encountering an unknown or not well known software, the distribution map together with the labeling provides a good first impression of the software's domain. Semantic clustering captures concepts regardless of class hierarchies, packages and other structures. One can, at a glance, see whether the software covers just a few or many different concepts, how these are distributed over the structure, and – due to the labeling – what they are about.

On detecting code duplication. As illustrated in the Moose example, the correlation matrix is useful to detect code duplication. Not on the level of repeated code fragments and string comparison [DRD99], but – as we work with lingustic data – high-level concept clones [MM01]: implementations of the same functionality that use different vocabularies. When developers lose the overview in a large project they tend to reinvent the wheel, since they are not aware that there is already an implementation somewhere. LSI can detect these kind of duplication because of its strength in identifying synonyms.

On the interactivity of the implementation. We implemented our approach in Hapax, a tool built on top of the Moose reengineering environment [DGLD05]. Figure 7.1 emphasizes the interactive nature of our tool.

On the left we see the main window of Hapax. On left part of the window is the correlation matrix visualization. On the right side of the window, there are three panels that are updated as the mouse moves over the matrix. The top two panels show the entity on the current row and the entity on the current column. The bottom panel shows the labels attached to the current cluster.

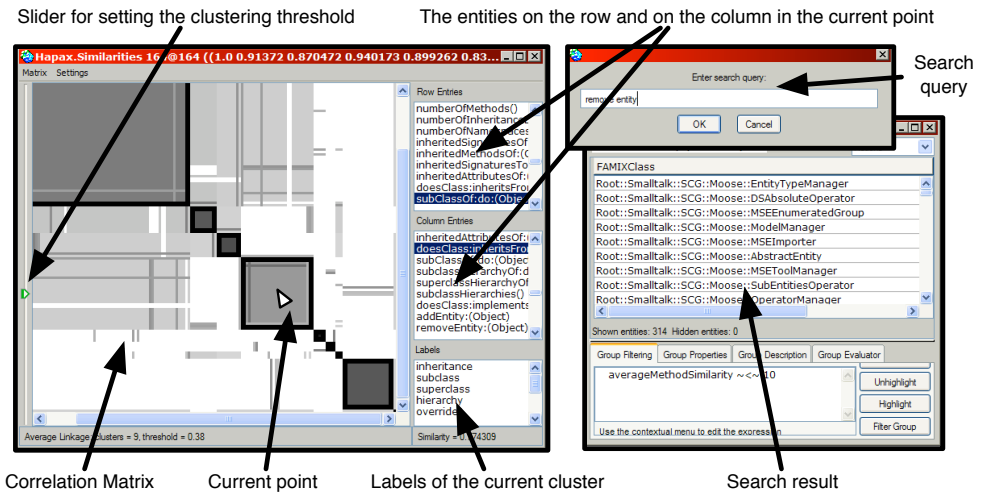

Figure 7.1: Hapax and Moose. To the left we show the main Hapax window. To the right we show how we can search for the entities relevant to a query string.

On the right side of the window there is a slider for setting the clustering threshold. When the slider is moved, the picture is redrawn with the new clusters. This feature allows one to explore different clustering configurations.

On the right side of the figure we show how we use LSI to also search over the entities in the system. The top window contains the search query and the result is shown in the below window with the group of the entities ordered by their relevancy to the query.

On using the approach as navigation aid. As semantic clustering returns a dendrogram, that is tree of clusters, we can use this to navigate the source code in a top-down fashion. We combined Hapax with *Softwarenaut*, an environment for the interactive, visual exploration of any hierarchical decomposition of a software system [LKGL05]. The user can interact with, and navigate the visualizations of the semantical clusters, aided by complementary lower level information about the properties and interconnections between the components of the clusters, see Figure 7.2.

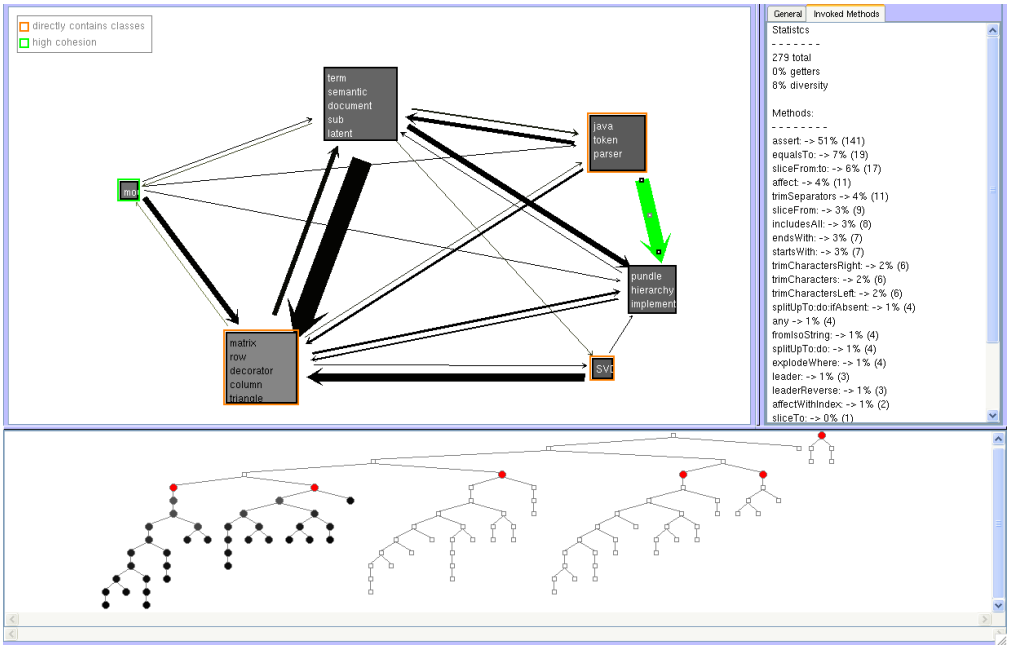

Figure 7.2: Hapax and Softwarenaut. On the left: the semantic clusters as boxes with arrows illustrating the number of invocations between them. On the right: the labels for the selected cluster. On the bottom: the dendrogram, a tree view of the clustering.

Chapter 8

Conclusions

"In real life, unlike in Shakespeare,
the sweetness of the rose depends upon the name it bears."
— Hubert H. Humphrey

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 exactly these parts of the source code.

This dissertation presented the use of *Semantic Clustering* to analyze the textual content of source code to recover domain concepts from the code itself [KDG05]. To identify the different concepts in the code, we apply Latent Semantic Indexing (LSI) and cluster the source artifacts according to the vocabulary of identifiers and comments. Each clusters represent a distinct domain concept. To find out what the concepts are about, we use LSI as search engine to retrieve the most relevant labels for the clusters. For each cluster, the labels are obtained by ranking and filtering the most similar terms [KDG06].

We use a shaded correlation matrix to visualize the identified concepts. It illustrates the *semantic similarity* between source artifacts and the par-

tition of the system into groups of software artifacts that implement the same domain concept; and it reveals *semantic links* between these clusters, that is single software artifacts which interconnect the domain concepts. We use a visualization based on the concept of a distribution map [DGK06] to show on the structure of the system (for example the package structure) how the different entities (for example classes) are touched by concepts.

We implemented our approach in a tool called Hapax [Kuh05]. Hapax is built on top of the Moose reengineering environment [DGLD05], We showed the generality of the approach by using the tool to analyze several case studies at different levels of abstraction and written in Java and Smalltalk [KDG05, KGG05, LKGL05, KDG06].

In the future we would like to investigate in more depth the relationship between the concepts and the structure. For example, we would like to compare the results of the semantic clustering with other types of clustering. Also, we would like to improve the labeling with other computer linguistic techniques.

We plan to investigate how semantic clustering can be useful for recovering the architecture. For example, in a layered architecture, each layer uses a specific vocabulary. Hence if we cluster classes based on semantic similarities and compare this with the structural packages we can detect classes that are placed in the wrong package. Furthermore, semantic clustering is able to detect the business domains which are orthogonal to the layers in the same way, as also each domain uses its own vocabulary.

On the other hand we would like to integrate an LSI engine in an IDE to support documentation and search.

The labeling, or more simply just a search with a single software artifact as query, can propose possible keywords to be used in the documentation or comment of a software artifact. Furthermore, when building an index with both the source code and its documentation as a separated document, we can detect bad or out-of-date comments. Hence our approach can be used to grade Javadoc like comments; this is similar to the essay grading [FLL99].

Since semantic clustering makes use of an index created based on LSI, we can simply use it to provide a search functionality that goes beyond

mere keyword matching, because LSI takes synonymy and polysemy[1] into account. This is most useful in large projects where a single developer can not know the whole project and its exact vocabulary. This prevents high-level concept clones before they get written, since a search query finds the desired implementation even if the query terms itself do not match exactly.

[1]Synonymy refers to different words having the same meaning and polysemy refers to the same word having different meanings.

List of Figures

2.1 The same image at different SVD approximations. From top left to bottom right: the original image with rank 200, an approximation with rank 20, rank 10, rank 5, rank 2 and finally rank 1. For the purpose of LSI, the rank 5 approximation would be sufficient. 11

2.2 On the left: An LSI-Space with terms and documents, similar elements are placed near each other. On the right: the grey cone is a search for documents related to term *point*, the red cone is a reverse-search for terms related to document #2. 12

4.1 From left to right: unordered correlation matrix, then sorted by similarity, then grouped by clusters, and finally including semantic links. 24

4.2 Hierarchical clustering yields a tree, called "dendrogram", that imposes both a sort order and a grouping on its leaf elements 26

4.3 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. 27

4.4 A sample *Correlation Matrix*: Cluster A and B are similar, cluster C is very cohesive, two elements in B are related to C. 28

4.5 Automatically retrieved labels describe the concepts. The labels were retrieved using the documents in a concept cluster as query to search the LSI space for related terms. . . . 30

5.1 The *Distribution Map* shows how semantic concepts are distributed over the structural modularization of a system. Boxes denote modules and software artifacts, colors refer to concepts. 32

5.2 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. 33
5.3 This *Distribution Map* illustrates the distribution of semantic concepts over packages and classes of the Outsight case-study. There are both encapsulated and cross-cutting concepts. 35
5.4 All semantic concepts of the Outsight case-study: the *top seven labels* column lists the most relevant terms of each concept, and *main module* column lists the package that contains most of its classes. 35

6.1 The case studies and their characteristics. 39
6.2 The correlation matrix of the Moose environment and four plug-ins. 40
6.3 The correlation matrix of the methods inside the MSEModel class. 43
6.4 The distribution map of the semantic clusters found in JBoss. 44
6.5 The labels of the semantic clusters of JBoss. 44
6.6 The distribution map of the semantic clusters found in Azureus. 46
6.7 Distribution Map of linguistic concepts over the packages of JEdit 31 parts, 394 elements and 9 properties). 48
6.8 Distribution Map of linguistic concepts over the packages of Ant (66 parts, 665 elements and 9 properties). 49

7.1 Hapax and Moose. To the left we show the main Hapax window. To the right we show how we can search for the entities relevant to a query string. 54
7.2 Hapax and Softwarenaut. On the left: the semantic clusters as boxes with arrows illustrating the number of invocations between them. On the right: the labels for the selected cluster. On the bottom: the dendrogram, a tree view of the clustering. 55

Bibliography

[ACC+02] 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.

[ACCL00] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 40–49, 2000.

[AL98] 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.

[BDO95] 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.

[Ber73] Jacques Bertin. *Sémiologie graphique*. Les Re-impressions des Editions de l'Ecole des Hautes Etudes En Sciences Sociales, 1973.

[Ber81] Jaques Bertin. *Graphics and Graphic Information Processing*. Walter de Gruyter, 1981.

[Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22:36–49, October 1989.

[Buc85] Chris Buckley. Implementation of the smart information re-

trieval system. Technical Report TR85-686, Cornell University, Ithaca, NY, USA, 1985.

[CHSDZ05] Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings of 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 135–144, 2005.

[CT99] 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.

[DDL+90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Analyzing the distribution of properties in software systems. In *Submitted to International Conference on Program Comprehension (ICPC 2006)*, 2006.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DN92] Susan T. Dumais and Jakob Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *Research and Development in Information Retrieval*, pages 233–244, 1992.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings of theInternational Conference on Software Maintenance (ICSM '99)*, pages 109–118. IEEE Computer Society, September 1999.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse.

FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Dum91] Susan T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23:229–236, 1991.

[DVRG+05] Serge Demeyer, Filip Van Rysselberghe, Tudor Gîrba, Jacek Ratzinger, , Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza, Matthias Rieger, Harald Gall, Michel Wermelinger, and Mohammad El-Ramly. The Lan-simulation: A Research and Teaching Example for Refactoring. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 123–131, Los Alamitos CA, 2005. IEEE Computer Society Press.

[EGK+01] Stephen Eick, Todd Graves, Alan Karr, J. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[FLL99] Peter Foltz, Darrell Laham, and Thomas Landauer. Automated essay scoring: Applications to educational technology. In *Proceedings World Conference on Educational Multimedia, Hypermedia and Telecommunications (EdMedia 1999)*, pages 939–944, 1999.

[FN87] William Frakes and Brian Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[FSvH03] Christiaan Fluit, Marta Sabou, and Frank van Harmelen. Supporting user tasks through visualisation of light-weight ontologies. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*. Springer-Verlag, 2003.

[HHDO03] Jane Huffman-Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Procerdings of 11th IEEE International Requierments Engineering Conference*, page 138, 2003.

[Jav] Java. http://java.sun.com/.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[KDG05] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. 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.

[KDG06] Adrian Kuhn, Stephane Ducasse, and Tudor Girba. Semantic clustering: Exploiting source code linguistic information. *Information and Software Technology*, submitted, 2006.

[KGG05] Adrian Kuhn, Orla Greevy, and Tudor Gîrba. Applying semantic analysis to feature execution traces. In *Proceedings of Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005)*, pages 48–53, November 2005.

[KGMI04] 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.

[Kuh05] Adrian Kuhn. Hapax – enriching reverse engineering with semantic clustering, November 2005.

[LD91] T. Landauer and S. Dumais. The latent semantic analysis theory of acquisition, induction, and representation of knowledge. In *Psychological Review*, volume 104/2, pages 211–240, 1991.

[LD05] Michele Lanza and Stéphane Ducasse. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

[LFOT04] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of 20th IEEE*

International Conference on Software Maintainance (ICSM 2004), pages 306–315, 2004.

[LKGL05] Mircea Lungu, Adrian Kuhn, Tudir Gîrba, and Michele Lanza. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 95–100, 2005.

[LPG02] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Sara Gradara. An approach to classify software maintenance requests. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 93–102, 2002.

[Luh58] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2:159–165, 1958.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[LY01] Yang Li and Hongji Yang. Simplicity: A key engineering concept for program understanding. In *proceddings of 9th International Workshop on Programm Comprehension*, 2001.

[MBK91] Yoëlle 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.

[Mer95] Dieter Merkl. Content-based software classification by self-organization. In *Proceddings of International Conference on Neural Networks (ICNN'95)*, volume II, pages 1086–1091, 1995.

[MM00] Jonathan I. Maletic and Andrian 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)*, pages 46–53, November 2000.

[MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, November 2001.

[MM03] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 125–135, May 2003.

[MMD93] P. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionist models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'93)*, volume 1, pages 1339–1345, 1993.

[MP05] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings Internationl Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

[Nak01] Preslav Nakov. Latent semantic analysis for german literature investigation. In *Proceedings of the International Conference, 7th Fuzzy Days on Computational Intelligence, Theory and Applications*, pages 834–841, London, UK, 2001. Springer-Verlag.

[oDRC+02] Johan Natt och Dag, Björn Regnell, Pär Carlshamre, Michael Andersson, and Joachim Karlsson. A feasibility study of automated natural language requirements analysis in market-driven development. *Requirements Engineering*, 7(1):20–33, 2002.

[Par94] David Lorge Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, Los Alamitos CA, 1994. IEEE Computer Society.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[SCHK+04] Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, and Chris DePalma. Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Proceedings of 7th International Workshop on Principles of Software Evolution*, pages 49–54, 2004.

[SR99] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, November/December 1999. Special Section: International Conference on Software Maintenance (ICSM'97).

[TH99] Vassilios Tzerpos and Rick Holt. MoJo: A distance metric for software clusterings. In *Proceedings Working Conference on Reverse Engineering (WCRE 1999)*, pages 187–195, Los Alamitos CA, 1999. IEEE Computer Society Press.

[Ton01] Paolo Tonella. Concept Analysis for Module Restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, April 2001.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 246–256, New York, May 1999. Association for Computing Machinery.

[WHH05] 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.

[Wig97] Theo Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 33–43. IEEE Computer Society Press, 1997.

[WSB+98] L. Wood, J. Sorensen, S. Byrne, R.S. Sutor, V. Apparao, S. Isaacs, G. Nicol, and M. Champion. *Document Object Model Specification DOM 1.0.* World Wide Web Consortium,

1998.

[Zip49] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology.* Addison-Wesley Press Inc., Cambridge 42, MA, USA, 1949.

[ZZL+04] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static non-interactive approach to feature location. In *Proceedings of 26th International Conference on Software Engineering (ICSM'04)*, pages 293–303, Edinburgh, UK, 2004.