Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 230–243
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Semantic clustering: Identifying topics in source code

Adrian Kuhn [a,*,1], Stéphane Ducasse [b,2], Tudor Gîrba [a,1]

[a] Software Composition Group, University of Berne, Switzerland
[b] Language and Software Evolution Group, LISTIC, Université de Savoie, France

Received 4 July 2006; accepted 25 October 2006
Available online 4 January 2007

Abstract

Many of the existing approaches in Software Comprehension focus on program structure or external documentation. However, by analyzing formal information the informal semantics contained in the vocabulary of source code are overlooked. To understand software as a whole, we need to enrich software analysis with the developer knowledge hidden in the code naming. This paper proposes the use of information retrieval to exploit linguistic information found in source code, such as identifier names and comments. We introduce *Semantic Clustering*, a technique based on Latent Semantic Indexing and clustering to group source artifacts that use similar vocabulary. We call these groups *semantic clusters* and we interpret them as *linguistic topics* that reveal the intention of the code. We compare the topics to each other, identify links between them, provide automatically retrieved labels, and use a visualization to illustrate how they are distributed over the system. Our approach is language independent as it works at the level of identifier names. To validate our approach we applied it on several case studies, two of which we present in this paper.

Note: Some of the visualizations presented make heavy use of colors. Please obtain a color copy of the article for better understanding.

Keywords: Reverse engineering; Clustering; Latent Semantic Indexing; Visualization

1. Introduction

Acquiring knowledge about a software system is one of the main activities in software reengineering, it is estimated that up to 60% of software maintenance is spent on comprehension [1]. This is because a lot of knowledge about the software system and its associated business domain is not captured in an explicit form. Most approaches that have been developed focus on program structure [2] or on external documentation [3,4]. However, there is another fundamental source of information: the developer knowledge contained in identifier names and source code comments.

> "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 [5]."

Languages are a means of communication, and programming languages are no different. Source code contains two levels of communication: human–machine communication through program instructions, and human to human communications through names of identifiers and comments. Let us consider a small code example:

* Corresponding author.
E-mail addresses: akuhn@iam.unibe.ch (A. Kuhn), sduca@unv-savoie.fr (S. Ducasse), girba@iam.unibe.ch (T. Gîrba).
[1] We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).
[2] We gratefully acknowledge the financial support of the french ANR for the project "Cook: Réarchitecturisation des applications à objets".

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.10.017

```
public boolean isMorning(int hours, int minutes, int seconds) {
    if (! isDate(hours, minutes, seconds))
        throw Exception("Invalid input: not a time value.");
    return hours < 12 && minutes < 60 && seconds < 60;
}
```

When we strip away all identifiers and comments, from the machine point of view the functionality remains the same, but for a human reader the meaning is obfuscated and almost impossible to figure out. In our example, retaining formal information only 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 < A && b < B && c < C;
}
```

When we keep only the informal information, the purpose of the code is still recognizable. In our example, retaining only the naming yields:

> is morning hours minutes seconds is date hours minutes seconds invalid time value hours 12 minutes 60 seconds 60

In this paper, we use information retrieval techniques to *derive topics from the vocabulary usage at the source code level*. Apart from external documentation, the location and use of source-code identifiers is the most frequently consulted source of information in software maintenance [6]. The objective of our work is to analyze software without taking into account any external documentation. In particular we aim at:

- *Providing a first impression of an unfamiliar software system*. A common pattern when encountering an unknown or not well known software for the first time is ''Read all the Code in One Hour'' [7]. Our objective is to support this task, and to provide a map with a survey of the system's most important topics and their location.
- *Revealing the developer knowledge hidden in identifiers*. In practice, it is not external documentation, but identifier names and comments where developers put their knowledge about a system. Thus, our objective is not to locate externally defined domain concepts, but rather to derive topics from the actual use of vocabulary in source code.
- *Enriching Software Analysis with informal information*. When analyzing formal information (e.g., structure and behavior) we get only half of the picture: a crucial source of information is missing, namely, the semantics contained in the vocabulary of source code. Our objective is to reveal components or aspects when, for example, planning a large-scale refactoring. Therefore, we analyze how the code naming compares to the code structure: What is the distribution of linguistic topics over a system's modularization? Are the topics well-encapsulated by the modules or do they cross-cut the structure?

Our approach is based on Latent Semantic Indexing (LSI), an information retrieval technique that locates linguistic topics in a set of documents [8,9]. We apply LSI to compute the linguistic similarity between source artifacts (e.g., packages, classes or methods) and cluster them according to their similarity. This clustering partitions the system into linguistic topics that represent groups of documents using similar vocabulary. To identify how the clusters are related to each other, we use a correlation matrix [10]. We employ LSI again to automatically label the clusters with their most relevant terms. And finally, to complete the picture, we use a map visualization to analyze the distribution of the concepts over the system's structure.

We implemented this approach in a tool called Hapax,[3] which is built on top of the Moose reengineering environment [11,12], and we apply the tool on several case studies, two of which are presented in this work: JEdit[4] and JBoss.[5]

This paper is based on our previous work, in which we first proposed semantic clustering [13]. The main contributions of the current paper are:

- *Topic distribution analysis*. In our previous work we introduced semantic clustering to detect linguistic topics given by parts of the system that use similar vocabulary. We complement the approach with the analysis of how topics are distributed over the system using a Distribution Map [14].
- *Improvement of the labeling algorithm*. One important feature of semantic clustering is the automatic labeling – i.e., given a cluster we retrieve the most relevant labels for it. We propose an improved algorithm that takes also the similarity to the whole system into account.
- *Case studies*. In our previous work, we showed the results of the clustering and labeling on different levels of abstraction on three case studies. In this paper we report on other two case studies.

Structure of the paper. In the next section we describe the LSI technique. In Section 3 we show how we use LSI to analyze the semantics of the system and how we can apply the analysis at different levels of abstraction. In Section 4 we present our approach to analyze the distribution of the semantic clusters over the structure of the system. In Section 5 we present the results of two case studies. We discuss the approach in Section 6. In Section 7 we outline the related work and Section 8 concludes and presents the future work.

2. Latent Semantic Indexing

As with most information retrieval techniques, Latent Semantic Indexing (LSI) is based on the vector space model

[3] The name is derived from the term *hapax legomenon*, that refers to a word occurring only once a given body of text.
[4] http://www.jedit.org/
[5] http://www.JBoss.org/

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 improves the basic vector space model by replacing the original term–document matrix with an approximation. This is done using singular value decomposition (SVD), a principal components analysis (PCA) technique originally used in signal processing to reduce noise while preserving the original signal. Assuming that the original term–document matrix is noisy (the 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 search engine covers a text corpus with millions of web pages, containing some ten thousands of terms, which is reduced to a vector space with 200–500 dimensions only. In Software Analysis, the number of documents is much smaller and we typically reduce the text corpus to 20–50 dimensions.

Even though search engines are the most common uses of LSI [15], there is a wide range of applications, such as automatic essay grading [16], automatic assignment of reviewers to submitted conference papers [17], cross-language search engines, thesauri, spell checkers and many more. In the field of software engineering LSI has been successfully applied to categorized source files [18] and open-source projects [19], detect high-level conceptual clones [20], recover links between external documentation and source code [21,22] and to compute the class cohesion [22]. 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 [23].

Fig. 1 schematically represents the LSI process. The document collection is modeled as a vector space. Each document is represented by the vector of its term occurrences, where terms are words appearing in the document. The term–document matrix A is a sparse matrix and represents the document vectors on the rows. This matrix is of size $n \times m$, where m is the number of documents and n the total number of terms over all documents. Each entry $a_{i,j}$ is the frequency of term t_i in document d_j. A geometric interpretation of the term–document matrix is as a set of document vectors occupying a vector space spanned by the terms. The similarity between documents is typically defined as the cosine or inner product between the corresponding vectors. Two documents are considered similar if their corresponding vectors point in the same direction.

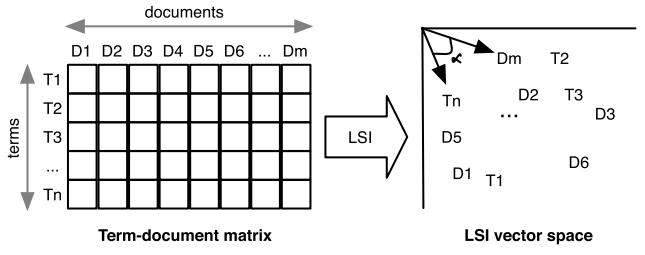

Fig. 1. LSI takes as input a set of documents and the terms occurrences, and returns as output a vector space containing all the terms and all the documents. The similarity between two items (i.e., terms or documents) is given by the angle between their corresponding vectors.

LSI starts with a raw term–document matrix, weighted by a weighting function to balance out very rare and very common terms. SVD is used to break down the vector space model into less dimensions. This algorithm preserves as much information as possible about the relative distances between the document vectors, while collapsing them into a much smaller set of dimensions.

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

As the rank is the number of linear-independent rows and columns of a matrix, the vector space spanned by A' is of dimension k only and much less complex than the initial space. When used for information retrieval, k is typically about 200–500, while n and m may go into millions. When used to analyze software on the other hand, k is typically about 20–50 with vocabulary and documents in the range of thousands only. And since A' is the best approximation of A under the least-square-error criterion, the similarity between documents is preserved, while in the same time mapping semantically related terms on one axis of the reduced vector space and thus taking into account synonymy and polysemy. In other words, the initial term–document matrix A is a table with term occurrences and by breaking it down to much less dimension the latent meaning *must* appear in A' since there is now much less space to encode the same information. Meaningless occurrence data is transformed into meaningful concept information.

3. Semantic clustering: Grouping source documents

The result of applying LSI is a vector space, based on which we can compute the similarity between both documents or terms. We use this similarity measurement to identify topics in the source code.

Fig. 2 illustrates the first three steps of the approach: preprocessing, applying LSI, and clustering. Furthermore, we retrieve the most relevant terms for each cluster and visualize the clustering on a 2D-map, thus in short the approach is:

(1) *Preprocessing the software system*. In Section 3.2, we show how we break the system into documents and

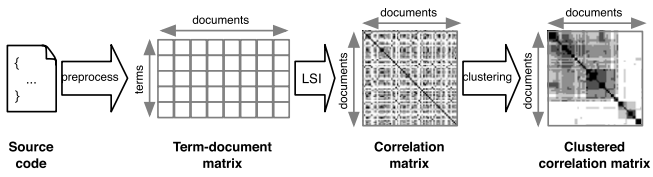

Fig. 2. Semantic clustering of software source code (e.g., classes and methods).

how we build a term–document matrix that contains the vocabulary usage of the system.

(2) *Applying Latent Semantic Indexing*. In Section 3.3 we use LSI to compute the similarities between source code documents and illustrate the result in a correlation matrix [10].

(3) *Identifying topics*. In Section 3.4 we cluster the documents based on their similarity, and we rearrange the correlation matrix. In Section 3.1 we discuss that each cluster is a *linguistic topic*.

(4) *Describing the topics with labels*. In Section 3.5 we use LSI again to retrieve for each cluster the top-*n* most relevant terms.

(5) *Comparing the topics to the structure*. In Section 4 we illustrate the distribution of topics over the system on a Distribution Map [14].

We want to emphasize that the primary contribution of our work is semantic clustering and the labeling. The visualization we describe are just used as a means to convey the results and are not original contributions of this paper.

3.1. On the relationship between concepts and semantic clustering

In this paper, we are interested in locating concepts in the source code. However, our concern is not to locate the implementation of externally defined domain concepts, but rather to derive the implemented topics from the vocabulary of source code. We tackle the problem of concept location on the very level of source code, where we apply information retrieval to analyze the use of words in source code documents. The clusters retrieved by our approach are not necessarily domain or application concepts, but rather code-oriented topics.

Other than with domain or application concepts in Software Analysis [24], it is not uncommon in information retrieval to derive *linguistic topics* from the distribution of words over a set of documents. This is in accordance with Wittgenstein who states that "*Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache – the meaning of a word is given by its use in language*" [25]. Unlike in classical philosophy, as for example Plato, there is no external definition of a word's meaning, but rather the meaning of a word is given by the relations it bears with other terms and sentences being used in the same context. Certainly, there is a congruence between the external definition of a word and its real meaning, since our external definitions are human-made as well, and thus, also part of the language game, but this congruence is never completely accurate.

In accordance with [5] we call our clusters *linguistic topics* since they are derived from language use. Certainly, some linguistic topics do map to the domain and others do map to application concepts, however, this mapping is never complete. There is no guarantee that semantic clustering locates all or even any externally defined domain concept. But nonetheless, our case studies show that semantic clustering is able to capture important domain and application concepts of a software system. Which does not come as a surprise, since it is well known that identifier names and comments are one of the most prominent places where developers put their knowledge about a system.

3.2. Preprocessing the software system

When we apply LSI to a software system we partition 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 packages or classes and methods. Other slicing solutions are possible as well, for example execution traces [26], or we can even use entire projects as documents and analyze a complete source repository [19].

To build the term–document matrix, we extract the vocabulary from the source code: we use both identifier names and the content of comments. Natural language text in comments is broken into words, whereas compound identifier names are split into parts. As most modern naming conventions use camel case, splitting identifiers is straightforward: for example *FooBar* becomes *foo* and *bar*.

We exclude common stopwords from the vocabulary, as they do not help to discriminate documents. In addition, if the first comment of a class contains a copyright disclaimer, we exclude it as well. To reduce words to their morphological root we apply a stemming algorithm: for example *entity* and *entities* both become *entiti* [27]. And finally, the term–document matrix is weighted with *tf-idf* to balance out the influence of very rare and very common terms [28].

When preprocessing object-oriented software systems we take the inheritance relationship into account as well. For example, when applying our approach on the level of classes, each class inherits some of the vocabulary of its superclass. If a method is defined only in the superclass we add its vocabulary to the current class. Per level of inheritance a weighting factor of $w = 0.5$ applies to the term occurrences, to balance out between the abstractness of high level definitions and concrete implementations.

3.3. Using Latent Semantic Indexing to build the similarity index

We use LSI to extract linguistic information from the source code, which results in an LSI-index with similarities

between source documents (i.e., packages, classes or methods). Based on the index we can determine the similarity between source code documents. Documents are more similar if they cover the same topic, terms are more similar if they denote related topics.

In the vector space model there is a vector for each document. For example, if we use methods as documents, there is a vector for each method and the cosine between these vectors denotes the semantic similarity between the methods. In general cosine values are in the $[-1, 1]$ range, however when using an LSI-index the cosine between its element never strays much below zero. This is since the LSI-index is derived from a term–document matrix that contains positive occurrence data only.

First matrix in Fig. 3. To visualize similarities between documents we map them to gray values: the darker, the more similar. The similarities between elements are arranged in a square matrix called *correlation matrix* or *dot plot*. Correlation matrix is a common visualization tool to analyze patterns in a set of entities [10]. Each dot $a_{i,j}$ denotes the similarity between element d_i and element d_j. Put in other words, the elements are arranged on the diagonal and the dots in the off-diagonal show the relationship between them.

3.4. Clustering: Ordering the correlation matrix

Without proper ordering the correlation matrix looks like television tuned to a dead channel. An unordered matrix does not reveal any patterns. An arbitrary ordering, such as for example the names of the elements, is generally as useful as random ordering [29]. Therefore, we cluster the matrix to put similar elements near each other and dissimilar elements far apart of each other.

A clustering algorithm groups similar elements together and aggregates them into clusters [30]. Hierarchical clustering creates a tree of nested clusters, called *dendrogram*, which has two features: breaking the tree at a given threshold groups the elements into clusters, and traversing the tree imposes a sort order upon its leaves. We use these two features to rearrange the matrix and to group the dots into rectangular areas.

Second and third matrix in Fig. 3. Each rectangle on the diagonal represents a semantic cluster: the size is given by the number of classes that belong to a topic, the color refers to the *semantic cohesion* [22] (i.e., the average similarity among its classes[6]). The color in the off-diagonal is the darker the more similar to clusters are, if it is white they are not similar at all. The position on the diagonal is ordered to make sure that similar topics are placed together.

[6] Based on the similarity $\mathrm{sim}(a, b)$ between elements, we define the similarity between cluster A and cluster B as $\frac{1}{|B|\times|A|}\sum\sum \mathrm{sim}(a_m, b_n)$ with $a \in A$ and $b \in B$ and in the same way the similarity between an element a_0 and a cluster B as $\frac{1}{|B|}\sum \mathrm{sim}(a_0, b_n)$ with $B \in B$.

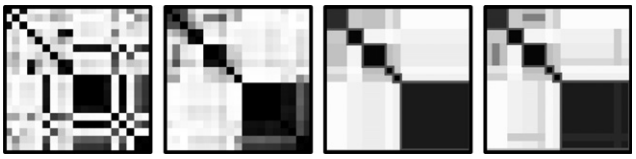

Fig. 3. From left to right: unordered correlation matrix, then sorted by similarity, then grouped by clusters, and finally including semantic links.

The clustering takes the focus of the visualization from similarity between elements to similarity between clusters. The tradeoff is, as with any abstraction, that some valuable detail information is lost. Our experiments showed that one-to-many relationships between an element and an entire cluster are valuable patterns.

Fourth matrix in Fig. 3. If the similarity between an element d_n from a cluster and another cluster differs more than a fix threshold from the average similarity between the two clusters, we plot d_n on top of the clustered matrix either as a bright line if d_n is less similar than average, or as a dark line if d_n is more similar than average. We call such a one-to-many relationship a *semantic link*, as it reveals an element that links from its own topic to another topic.

Fig. 4 illustrates an example of a semantic link. The relationship between clusters A and B is represented by the rectangle found at the intersection of the two clusters. The semantic link can be identified by a horizontal (or vertical) line that is darker than the rest of the rectangle. In our example, we find such a semantic link between two clusters. Please note that the presence of a link does not mean that the element in A should belong to B, it only means that the element in A is more similar to B then the other elements in A.

3.5. Labeling the clusters

With the clustering we partitioned the source documents by their vocabulary, but we do not know about the actual vocabulary that connects these documents together. In other words, what are the most important terms for each cluster? In this section, we use LSI to retrieve for each cluster the top-n list with its most relevant terms. We use these lists to label the topics.

The labeling works as follows. As we already have an LSI-index at hand, we use it as a search engine [15]. We reverse the usual search process where a search query of terms is used to find documents, and instead, we use the

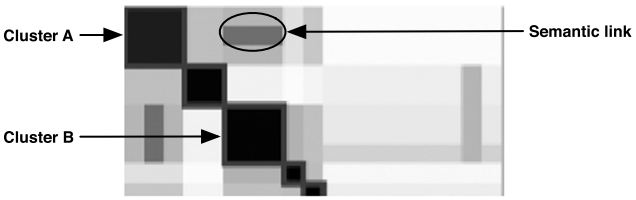

Fig. 4. Illustration of a semantic link. A semantic link is a one-to-many relation: A document in cluster A that is more similar than its siblings in A to cluster B.

documents in a cluster as search query to find the most similar terms. To label a cluster, we take the top-n most similar terms, using a top-7 list provides a useful labeling for most case studies.

To compute the relevance of a term, we compare the similarity to the current cluster with the similarity to all other clusters. This raises better results than just retrieving the top most similar terms [13]. Common terms, as for example the names of custom data structures and utility classes, are often highly similar to many clusters. Thus, these terms would pollute the top-n lists with non-discriminating labels if using plain similarity only, whereas subtracting the average lowers the ranking of such common terms. The following formula ranks high those terms that are very similar to the current cluster but not common to all other clusters.

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

Term t_0 is relevant to the current cluster A_0, if it has a high similarity to the current cluster A_0 but not to the remaining clusters $A_n \in \mathscr{A}$. Given the similarity between a term t and a cluster A as $\mathrm{sim}(t, A)$, we define the relevance of term t_0 according to cluster A_0 as given above.

Another solution to avoid such terms, is to include the common terms into a custom stopword list. However, this cannot be done without prior knowledge about the system, which is more work and contradicts our objectives. Furthermore, this ranking formula is much smoother than the strict opt-out logic of a stopword list.

4. Analyzing the distribution of semantic clusters

The semantic clusters help us grasp the topics 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 topics distributed over the system?

To answer this question, we use a Distribution Map [31,14]. A Distribution Map visualizes the distribution of properties over system parts, i.e., a set of entities. In this paper, we visualize packages and their classes, and color these classes according to the semantic cluster to which they belong.

For example, in Fig. 5 we show an example of a Distribution Map representing 5 packages, 37 classes and 4 semantic clusters. Each package is represented by a rectangle, which includes classes represented as small squares. Each class is colored by the semantic cluster to which it belongs.

Using the Distribution Map visualization we correlate linguistic information with structural information. 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, topics that correspond to the structure as well as topics that cross-cut it. Applying this visualization on several case studies, we identified the following patterns:

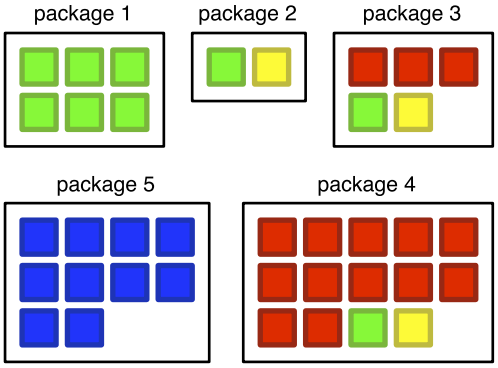

Fig. 5. Example of a Distribution Map.

- *Well-encapsulated topic* – if a topic corresponds to system parts, we call this a *well-encapsulated topic*. Such a topic is spread over one or multiple parts and includes almost all source code within those parts. If a well-encapsulated topic covers only one part we speak of a *solitary topic*.
- *Cross-cutting topic* – if a topic is orthogonal to system parts, we call this a *cross-cutting topic*. Such a topic spreads across multiple parts, but includes only one or very few elements within each parts. As linguistic information and structure are independent of each other, cross-cutting identifiers or names do not constitute a design flaw. Whether a cross-cutting topic has to be considered a design smell or not depends on the particular circumstances. Consider for example the popular three-tier architecture: It separates *accessing, processing and presenting data* into three layers; where application specific topics – such as e.g., *accounts, transactions or customers* – are deliberately designated to cross-cut the layers. That is, it emphasizes on the separation of those three topics and deliberately designates the others as cross-cutting concerns.
- *Octopus topic* – if a topic dominates one part, as a solitary does, but also spreads across other parts, as a cross-cutter does, we call this an *octopus topic*. Consider for example a framework or a library: there is a core part with the implementation and scattered across other parts there is source code that plug into the core, and hence use the same vocabulary as the core.
- *Black Sheep topic* – if there is a topic that consists only of one or a few separate source documents, we call this a *black sheep*. Each black sheep deserves closer inspection, as these documents are sometimes a severe design smell. Yet as often, a black sheep is just an unrelated helper class and thus not similar enough to any other topic of the system.

5. Case studies

To show evidence of the usefulness of our approach for software comprehension, in this section we apply it on two

case studies. Due to space limitations, only the first case study is presented in full length.

First, we exemplify each step of the approach and discuss its findings in the case of JEdit, a text editor written in Java. This case study is presented in full length. Secondly, we present JBoss, an application-server written in Java, which includes interesting anomalies in its vocabulary.

Fig. 6[7] summarizes the problem size of each case study. It lists for each case study: (lang) the language of the source code, (type) the granularity of documents, (docs) the number of documents, (terms) the number of terms, (parts) the number of found topics, (links) the number of found semantic links, (rank) the dimension of the LSI-index, and (sim) the threshold of the clustering.

5.1. On the calibration of parameters and thresholds

Our approach depends on several parameters, which may be difficult too choose for someone not familiar with the underlying technologies. In this section, we present all parameters, discuss their calibration and share our experience gained when performing case studies using the Hapax tool.

Weighting the term–document matrix. To balance out the influence of very rare and very common terms, it is common in information retrieval to weight the occurrence values. The most common weighting scheme is *tf-idf*, which we also use in the case studies, others are entropy or logarithmic weighting [28].

When experimenting with different weighting schemes, we observed that the choice of the weighting scheme has a considerable effect on the similarity values, depending on the weighting the distance within the complete text corpus becomes more compact or more loose [33]. Depending on the choice of the weighting scheme, the similarity thresholds may differ significantly: as a rule of thumb, using logarithmic weighting and a similarity threshold of $\delta = 0.75$ is roughly equivalent to a threshold of $\delta = 0.5$ with *tf-idf* weighting [34].

Dimensionality of the LSI-space. As explained in Section 2, LSI replaces the term–document matrix with a low-rank approximation. When working with natural language text corpora that contain millions of documents and some ten thousands of terms, most authors suggest to use an approximation between rank 200 and 500. In Software Analysis the number of documents is much smaller, such that even ranks as low as 10 or 25 dimensions yield valuable results. Our tool uses rank $r = (m * n)^{0.2}$ by default for an $m \times n$-dimensional text corpus, and allows customization.

Choice of clustering algorithm. There is a rich literature on different clustering algorithms [30]. We performed a series of experiments using different algorithms, however as we cannot compare our results against an *a priori* known partition, we cannot measure recall in hard numbers and have to rely on human judgment of the results. Therefore, we decided to use a hierarchical *average-linkage* clustering as it is a common standard algorithm. Further studies on the choice of clustering algorithm are open for future work.

Breaking the dendrogram into clusters. Hierarchical clustering uses a threshold to break the dendrogram, which is the tree of all possible clusters, into a fix partition. Depending on the objective, we break it either into a fixed number of clusters (e.g., for the Distribution Map, where the number of colors is constrained) or at a given threshold (e.g., for the correlation matrix). In the user interface of the Hapax tool, there is a slider for the threshold such that we can immediately observe the effect on both correlation matrix and Distribution Map interactively.

Detecting semantic links. As explained in Section 3.4, a semantics link is an element in cluster A with a different similarity to cluster B than average. Typically we use a threshold of $\delta = \pm 20\%$ to decide this, however, our tool supports fixed thresholds and selecting the top-n link as well. When detecting traceability-links, a problem which is closely related to semantic links, this has been proven as the best out if these three strategies [21].

5.2. Semantic clustering applied on JEdit

We exemplify our approach at the case of JEdit, an open-source Text editor written in Java. The case study contains 394 classes and uses a vocabulary of 1603 distinct terms. We reduced the text corpus to an LSI-space with rank $r = 15$ and clustered it with a threshold of $\delta = 0.5$ (the choice of parameters is discussed in Section 5.1).

In Fig 7, we see nine clusters with a size of (from top right to bottom left) 116, 63, 26, 10, 68, 10, 12, 80, and 9 classes. The system is divided into four zones: (zone 1) the large cluster in the top left, (zone 2) two medium sized and a small clusters, (zone 3) a large cluster and two small clusters, and (zone 4) a large and a small cluster. The two zones in the middle that are both similar to the first zone but not to each other, and the fourth zone is not similar to any zone.

In fact, there is a limited area of similarity between zones 2 and 3. We will later on identify the two counterparts as topics Pink and Cyan, which are related to text buffers and regular expression respectively. These two topics share some of their labels (i.e., start, end, length and count), however they are clustered separately since LSI does more than just keyword matching, LSI takes the context of term usage into account as well, that is the co-location of terms with other terms.

This is a common pattern that we often encountered during our experiments: zone 1 is the core of system with domain-specific implementation, zones 2 and 3 are facilities closely related to the core, and zone 4 is an unrelated component or even a third-party library. However, so far this is just an educated guess and therefore we will have a look at the labels next.

[7] The Moose case study in [13] did not use stemming to preproces the text corpus, hence the large vocabulary.

Case Study	language	type	docs	terms	parts	links	rank	sim
Ant	Java	*Classes*	665	1787	9	–	17	0.4
Azureus	Java	*Classes*	2184	1980	14	–	22	0.4
JEdit	Java	*Classes*	394	1603	9	53	17	0.5
JBoss	Java	*Classes*	660	1379	10	–	16	0.5
Moose [7]	Smalltalk	*Classes*	726	11785	–	137	27	–
MSEModel	Smalltalk	*Methods*	4324	2600	–	–	32	0.75
Outsight	Java	*Classes*	223	774	10	–	12	0.5

Fig. 6. The statistics of sample case studies, JEdit and JBoss are discussed in this work, for the other studies please refer to our previous work [13,32].

Fig. 8 lists for each cluster the top-7 most relevant labels, ordered by relevance. The labels provide a good description of the clusters and the tell same story as the correlation matrix before. We verified the labels and topics by looking at the actual classes within each cluster.

- Zone 1: topic Red implements the very domain of the system: files and users, and a user can load, edit and save files.
- Zone 2: topic Green and Magenta implement the user interface, and topic Pink implements text buffers.
- Zone 3: topic Cyan is about regular expressions, topic Yellow provides XML support and topic DarkGreen is about TAR archives.
- Zone 4: topic Blue and Orange are the BeanShell scripting framework, a third-party library.

All these labels are terms taken from the vocabulary of the source code and as such they do not always describe the topics in generic terms. For example, event though JEdit is a text-editor, the term *text-editor* is not used on the source code level. The same applies for topic Cyan, where the term *regular expression* does not shop up in the labels.

Next are the semantic links, these are the thin lines on the correlation matrix. If a line is thicker than one pixel, this means that there are two or more similar classes which are each a semantic links. The analysis detected 53 links, below we list as a selection all links related to the scripting topic (these are the link annotated on Fig. 7). For each link we list the class names and the direction:

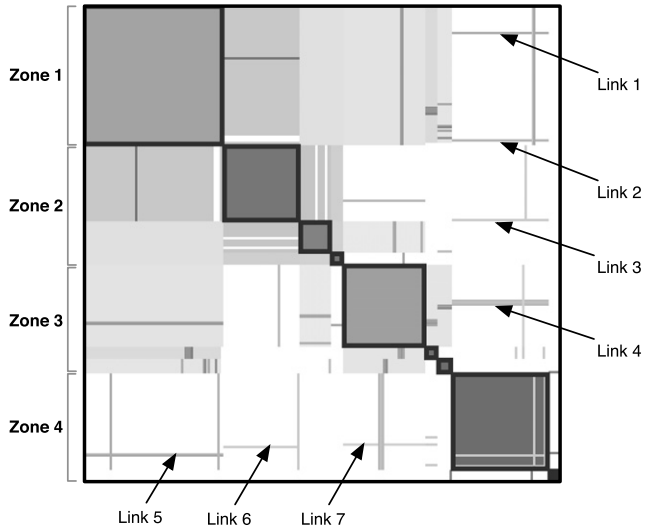

Fig. 7. The correlation matrix of JEdit.

- Link 1: *AppelScriptHandler* and *BeanShell* link from topic Red to scripting.
- Link 2: *Remote* and *ConsoleInterface* link from topic Red to scripting.
- Link 3: *VFSDirectoryEntryTable* and *JThis* link from the UI topic to scripting.
- Link 4: *ParseException*, *Token TokenMgrError* and *BSHLiteral* link from the regular expressions topic to scripting.
- Link 5: *ClassGeneratorUtil* and *ClasspathError* link from scripting to topic Red.
- Link 6: *ReflectManager* and *Modifiers* link from scripting to the UI topic.
- Link 7: *NameSource* and *SimpleNode* link from scripting to the regular expressions and the XML topic.

Evaluating semantic links is similar to evaluating traceability links [21]. We inspected the source code of each link. Some of them are false positives, for example links 4 and 5 are based on obviously unrelated use of the same identifiers. On the other hand, link 1 points to the main connector between core and scripting library and link 7 reveals a high-level clone, the implementation of the same datatype in three places: XML uses nodes, and both BSH and regular expressions implement their own AST node.

Fig. 9 shows the distribution of topics over the package structure of JEdit. The large boxes are the packages (the text above is the package name), the squares are classes and the colors correspond to topics (the colors are the same as on Fig. 8).

For example, in Fig. 9 the large box on the right represents the package named *bsh*, it contains over 80 classes and most of these classes implement the topic referred to by Blue. The package boxes are ordered by their similarity, such that related packages are placed near to each other.

Topic Red, the largest cluster, shows which parts of the system belong to the core and which do not. Based on the

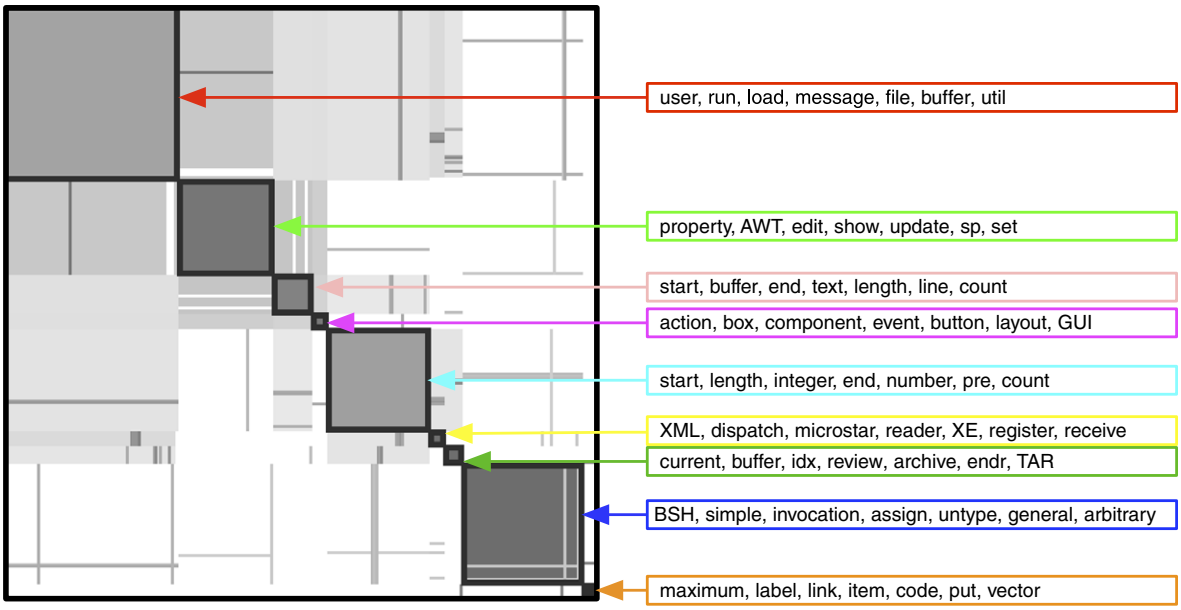

Fig. 8. The semantic clusters of JEdit and their labels.

ordering of the packages, we can conclude that the two UI topics (e.g., Green and Yellow) are more closely related to the core than for example topic Cyan, which implements regular expressions.

The three most well-encapsulated topics (e.g., Orange, Blue and Cyan) implement separated topics such as scripting and regular expressions. Topic Yellow and Pink crosscut the system: Yellow implements dockable windows, a custom GUI-feature, and Pink is about handling text buffers. These two topics are good candidates for a closer inspection, since we might want to refactor them into packages of their own.

5.3. First impression of JBoss: Distribution map and labels

This case study presents the outline of JBoss, an application-server written in Java. We applied semantic clustering and partitioned the system into 10 topics. The system is divided into one large cluster (colored in red), which implements the core of the server, and nine smaller clusters. Most of the small clusters implement different services and protocols provided by the application server.

The Distribution Map is illustrated in Fig. 10, and the top-7 labels are listed in figure Fig. 11 in order of relevance. This is the same setup as in the first case study, except the correlation matrix and semantic links are left out due to space restrictions. We verified the clustering by looking the source code, and present the results as follows.

Topic Red is the largest cluster and implements the core functionality of the system: is labeled with terms such as *invocation, interceptor, proxy* and *share*. Related to that, topic Cyan implements the deployment of JAR archives.

The most well-encapsulated topics are DarkGreen, Orange, Green and Blue. The first three are placed apart

Fig. 9. The Distribution Map of the semantic clusters over the package structure of JEdit.

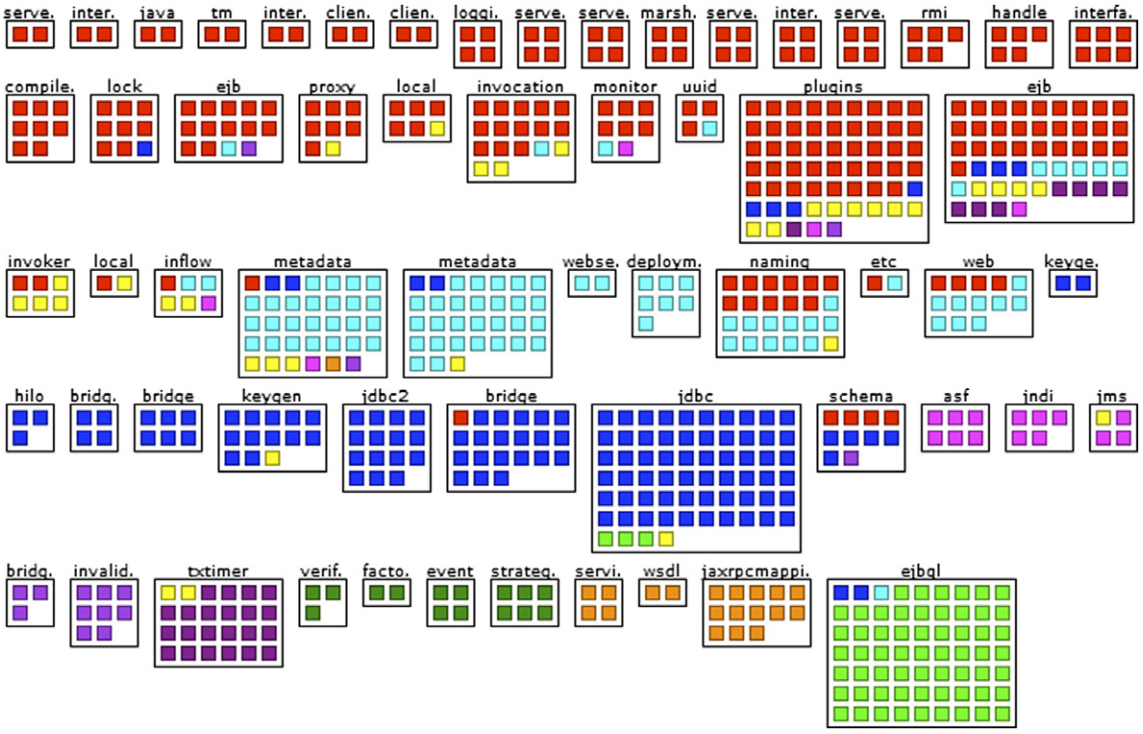

Fig. 10. The Distribution Map of the semantic clusters over the package structure of JBoss.

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

Fig. 11. The labels of the semantic clusters of JBoss.

from Red, whereas Blue has outliers in the red core packages. The labels and package names (which are printed above the package boxes in the Distribution Map) show that DarkGreen is a bean verifier, that Orange implements JAX-RPC and WDSL (e.g., web-services), that Green implements an SQL parser and that Blue provides JDBC (e.g., database access) support. These are all important topics of an application server.

The most cross-cutting topic is Yellow, it spreads across half of the system. The labels reveal that this the security aspect of JBoss, which is reasonable as security is an important feature within a server architecture.

Noteworthy is the label *loubyansky* in topic Blue, it is the name of a developer. Based on the fact that his name appears as one of the labels, we assume that he is the main developers of that part of the system. Further investigation proved this to be true.

Noteworthy as well are the labels of topic DarkGreen, 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 *package* statement of a Java class. In the case of topic DarkGreen this heuristic failed: the source files contained another licence within the body of the class. However, repeating the same case study with an improved preprocessing resulted in nearly the same clustering and labeled this cluster as RMI component: *event, receiver, RMI, RMIiop, iiop, RMIidl, and idl.*

The topics extracted from the source code can help improving the comprehension. If a maintainer is seeking information, semantic clustering helps in identifying the related code. This is similar to the use of a search engine, for example if the web-service interface has to be changed, the maintainer can immediately look at the Orange concept, and identify the related classes. Much in the same way, if one has to maintain the database interface, he looks at the Blue concept.

6. Discussion

In this section, we evaluate and discuss success criteria, strengths and limitations of the proposed approach. We discuss how the approach stands and fails with the quality of the identifier naming. Furthermore, we discuss the relation between linguistic topics and domain or application concepts.

6.1. On the quality of identifier names

In the same way as structural analysis depends on correct syntax, semantic analysis is sensitive to the quality of the naming. Since we derive our topics solely based on the use of identifier names and comments, it does not come as a surprise that our approach stands and fails with the quality of the source code naming.

Our results are not generalizable to any software system, a good naming convention and well chosen identifiers yields best results, whereas bad naming (i.e., too generic names, arbitrary names or cryptic abbreviations) is one of the main threats to external validation. The vocabulary of the case studies presented in this work is of good quality, however, when performing other case studies we learned of different facets that affect the outcome, these are:

On the use of naming conventions. Source following state-of-the-art naming conventions, as for example the Java Naming Convention, is easy to preprocess. In case of legacy code that uses other naming conventions (e.g., the famous Hungarian Notation) or even none at all, other algorithms and heuristics are to be applied [35,36].

On generic or arbitrary named identifiers. However, even the best preprocessing cannot guess the meaning of variables which are just named *temp* or *a*, *b* and *c*. If the developers did not name the identifiers with care, our approach fails, since the developer knowledge is missing. Due to the strength of LSI in detecting synonymy and polysemy, our approach can deal with a certain amount of such ambiguous or even completely wrong named identifiers – but if a majority of identifiers in system is badly chosen, the approach fails.

On abbreviated identifier names. Abbreviated identifiers are commonly found in legacy code, since early programming languages often restrict the discrimination of identifier names to the first few letters. But unlike generic names, abbreviations affect the labeling only and do not threat our approach as whole. This might come as a surprise, but since LSI is solely based on analyzing the statistical distribution of terms across the document set, it is not relevant whether identifiers are consistently written out or consistently abbreviated.

However, if the labeling task comes up with terms such as *pma*, *tcm*, *IPFWDIF* or *sccpsn* this does not tell a human reader much about the system. These terms are examples taken from a large industry case study, which is not included in this paper, where about a third of all identifiers where abbreviations. In this case the labeling was completely useless. Please refer to [36] for approaches on how to recover abbreviations.

On the size of the vocabulary. The vocabulary of source code is very small, smaller than that of a natural language text corpus. Intuitively explained: LSI is like a child learning language. In the same way as a human with a vocabulary of 2000 terms is less eloquent and knowledgeable than a human with a vocabulary of 20,000 terms, LSI performs better the larger the vocabulary. Whereas, the smaller the vocabulary the stronger the effect of missing or incorrect terms. In fact, LSI has been proven a valid model of the way children acquire language [23].

On the size of documents. 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 numbers are higher since comments are human-readable text. This is one of the rationales why LSI does not perform as accurate on source code as on natural language text [21], however the results are of sufficient quality.

On the combination of LSI with morphological analysis. Even tough the benefits of stemming are not without controversy, we apply it as part of the preprocessing step [37]. Our rational is: analyzing a software system at the level of methods is very sensitive to the quality of input, as the small document size threatens the success of LSI. Considering these circumstances, we decided to rely on stemming as it is well known that the naming of identifiers often includes the same term in singular and plurals: for example *setProperty* and *getAllProperties* or *addChild* and *getChildren*.

6.2. On using semantic clustering for topic identification

One of our objectives is to compare linguistic topics to domain and application concepts [24]. As discussed in Section 3.1, we derive linguistic topics from the vocabulary usage of source code instead from external definitions. In this section, we clarify some questions concerning the relation between derived topics and externally defined concepts.

On missing vocabulary and ontologies. Often the externally defined concepts are not captured by the labeling. The rational for this is as follows. Consider 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 text-editor concepts with these two terms, as a more generic term is missing. As our

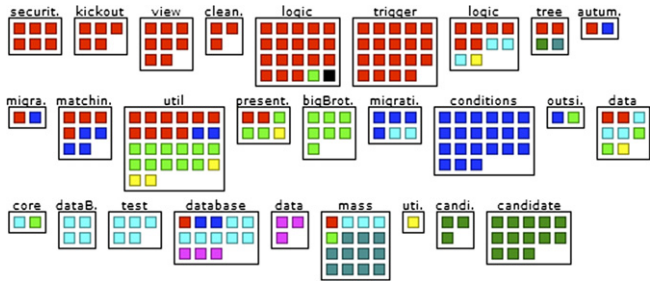

Fig. 12. The Distribution Map of Outsight, a webbased job portal application [32].

approach is not based on an ontological database, its vocabulary is limited to the terms found in source code and if terms are not used our approach will not find accurate labels. We suggest to use ontologies (i.e., WordNet) to improve the results in these cases.

On the congruence between topics and domain. When starting this work, one of our hypotheses was that semantic clustering will reveal a system's domain semantics. But our experiments disproved this hypothesis: most linguistic topics are application concepts or architectural components, such as layers. In many experiments, our approach partitioned the system into one (or sometimes two) large domain-specific part and up to a dozen domain-independent parts, such as for example input/output or data storage facilities. Consider for example the application in Fig. 12, it is divided into nine parts as follows:

Only one topic out of nine concepts is about the system's domain: job exchange. Topic Red includes the complete domain of the system: that is users, companies and CVs. Whereas all other topics are application specific components: topic Blue is a CV search engine, topic DarkGreen implements PDF generation, topic Green is text and file handling, topic Cyan and Magenta provide access to the database, and topic DarkCyan is a testing and debugging facility. Additionally the cross-cutting topic Yellow bundles high-level clones related to time and timestamps.

On the congruence between topics and packages. In section Section 4 we discussed the relation between topics and packages. Considering again the case study in Fig. 12 as an example, we find occurrences of all four patterns: Topic DarkGreen for example is well-encapsulated, whereas topic Yellow cross-cuts the application. Then there is topic Blue, which is an octopus with the *conditions* package as its body and tentacles reaching into six other packages, and finally we have in the *logic* package an instance of a black sheep.

On source documents that are related to more than one topic. If we want to analyze how the topics are spread across some type of documents (e.g., packages, classes or methods) we have to break the system into documents one level below the target level. For example, if we want to analyze the topic distribution over packages, we break the system into classes and analyze how the topics of classes are spread over the packages.

7. Related work

The use of information retrieval techniques for reverse engineering dates back to the late eighties. Frakes and Nejmeh proposed to apply them on source code as if it would be a natural language text corpus [38]. They applied an IR system based on keyword matching, which allowed to perform simple searches using wildcards and set expressions.

Antoniol et al. have published a series of papers on recovering code to documentation traceability [39,4]. They use information retrieval as well, however with another approach. They rely on external documentation as text corpus, then they query the documentation with identifiers taken from source code to get matching external documents.

Maletic and Marcus were the first to propose using LSI to analyze software systems. In a first work they categorized the source files of the Mosaic web browser and presented in several follow-ups other applications of LSI in software analysis [18]. Their work is a precursor of our work, as they proved that LSI is usable technique to compare software source documents. They apply a minimal-spanning-tree clustering and report on class names and average similarity of selected clusters. We broaden the approach by providing a visual notation that gives an overview of all the clusters and their relationships, and we provide the automatic labeling that takes the entire vocabulary into account and not only the class names.

LSI was also used in other related areas: Marcus and Maletic used LSI to detect high-level conceptual clones, that is they go beyond just string based clone detection using the LSI capability to spot similar terms [20]. They select a known implementation of an abstract datatype, and manually investigate all similar source documents to find high-level concept clones. The same authors also used LSI to recover links between external documentation and source code by querying the source code with queries from documentation [40].

Kawaguchi et al. used LSI to categorize software systems in open-source software repositories [19]. Their approach uses the same techniques as ours, but with a different set up and other objectives. They present a tool that categorizes software projects in a source repository farm, that is they use entire software systems as the documents of their LSI space. They use clustering to provide an overlapping categorizations of software, whereas we use clustering to partition the software into distinct topics. They use a visualization of they results with the objective to navigate among categorizations and projects, similar to the Softwarenaut tool [41], whereas we use visualizations to present an overview, including all documents and the complete partition, at one glance.

Marcus et al. employed LSI to detect concepts in the code [9]. They used the LSI as a search engine and searched in the code the concepts formulated as queries. Their work is about concept location of externally defined concepts, whereas we derive our concepts from the vocabulary usage

on the source-code level. Their article also gives a good overview of the related work. Marcus et al. also use LSI to compute the cohesion of a class based on the semantic similarity of its methods [22]. In our work, we extend this approach and illustrate on the correlation matrix both, the semantic similarity within a cluster and the semantic similarity between clusters.

De Lucia et al. introduce strategies to improve LSI-based traceability detection [21]. They use three techniques of link classification: taking the top-*n* search results, using a fix threshold or a variable threshold. Furthermore, they create separate LSI spaces for different document categories and observe better results that way, with best results on pure natural language spaces. Lormans and Deursen present two additional links classification strategies [42], and discuss open research questions in traceability link recovery.

Di Lucca et al. also focus on external documentation, doing automatic assignment of maintenance requests to teams [43]. They compare approaches based on pattern matching and clustering to information retrieval techniques, of which clustering performs better.

Huffman-Hayes et al. compare the results of several information retrieval techniques in recovering links between document and source code to the results of a senior engineer [44]. The results suggest that automatic recovery performs better than human analysis, both in terms of precision and recall and with comparable signal-to-noise ratio. In accordance with these findings, we automate the "Read all the Code in One Hour" pattern using information retrieval techniques.

Čubranić et al. build a searchable database with artifacts related to a software system, both source code and external documentation [45]. They use a structured meta model, which relates bug reports, news messages, external documentation and source files to each other. Their goal is the support software engineers, especially those new to a project, with a searchable database of what they call "group memory". To search the database they use information retrieval, however they do not apply LSI and use a plain vector space model only. They implemented their approach in an eclipse plug-in called Hipikat.

8. Conclusions

Source code bears the semantics of an application in the names of identifiers and comments. In this paper we present our approach to retrieve the topics present in the source code vocabulary to support program comprehension. We introduce semantic clustering, a technique based on Latent Semantic Indexing and clustering to group source documents that use similar vocabulary. We call these groups semantic clusters and we interpret them as linguistic topics that reveal the intention of the code. As compared to our previous approach, we go a step forward and use Distribution Maps to illustrate how the semantic clusters are distributed over the system.

We applied the approach on several case studies with different characteristics, two of which are presented in this paper. The case studies give evidence that our approach provides a useful first impression of an unfamiliar system, and that we reveal valuable developer knowledge. The Distribution Map together with the labeling provides a good first impression of the software's domain. Semantic clustering captures topics regardless of class hierarchies, packages and other structures. One can, at a glance, see whether the software covers just a few or many different topics, how these are distributed over the structure, and – due to the labeling – what they are about.

When starting this work, one of our hypotheses was that semantic clustering would reveal a systems domain semantics. However, our experiments showed that most linguistic topics relate to application concepts or architectural components. Usually, our approach partitions a system into one (or sometimes two) large domain-specific clusters and up to a dozen domain-independent clusters. As part of our future work, we plan to investigate more closely the relationship between linguistic topics and both domain and application concepts.

In the future, we would also like to investigate in more depth recall and precision of the approach. For example, we would like to compare the results of the semantic clustering with other types of clustering. Furthermore, we would like to improve the labeling with other computer linguistic techniques.

References

[1] A. Abran, P. Bourque, R. Dupuis, L. Tripp, Guide to the software engineering body of knowledge (ironman version), Tech. rep., IEEE Computer Society, 2004.
[2] S. Ducasse, M. Lanza, The class blueprint: Visually supporting the understanding of classes, IEEE Transactions on Software Engineering 31 (1) (2005) 75–90.
[3] Y.S. Maarek, D.M. Berry, G.E. Kaiser, An information retrieval approach for automatically constructing software libraries, IEEE Transactions on Software Engineering 17 (8) (1991) 800–813.
[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE Transactions on Software Engineering 28 (10) (2002) 970–983.
[5] T.J. Biggerstaff, Design recovery for maintenance and reuse, IEEE Computer 22 (1989) 36–49.
[6] J. Koskinen, A. Salminen, J. Paakki, Hypertext support for the information needs of software maintainers, Journal on Software Maintenance Evolution: Research and Practice 16 (3) (2004) 187–215.
[7] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufmann, Los Altos, CA, 2002.
[8] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, Journal of the American Society of Information Science 41 (6) (1990) 391–407.
[9] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, An information retrieval approach to concept location in source code, in: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), 2004, pp. 214–223.
[10] R.L. Ling, A computer generated aid for cluster analysis, Communications of ACM 16 (6) (1973) 355–361.
[11] S. Ducasse, T. Gîrba, M. Lanza, S. Demeyer, Moose: a collaborative and extensible reengineering environment, in: Tools for Software

Maintenance and Reengineering, RCOST Software Technology Series, Franco Angeli, Milano, 2005, pp. 55–71.
[12] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), ACM Press, New York, NY, 2005, pp. 1–10, invited paper.
[13] A. Kuhn, S. Ducasse, T. Gîrba, Enriching reverse engineering with semantic clustering, in: Proceedings of the Working Conference on Reverse Engineering (WCRE 2005), IEEE Computer Society Press, Los Alamitos, CA, 2005, pp. 113–122.
[14] S. Ducasse, T. Gîrba, A. Kuhn, Distribution map, in: Proceedings of the 22nd IEE International Conference on Software Maintenance (ICSM 2006), 2006, pp. 203–212.
[15] M.W. Berry, S.T. Dumais, G.W. O'Brien, Using linear algebra for intelligent information retrieval, SIAM Review 37 (4) (1995) 573–597.
[16] P. Foltz, D. Laham, T. Landauer, Automated essay scoring: Applications to educational technology, in: Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications (EdMedia 1999), 1999, pp. 939–944.
[17] S.T. Dumais, J. Nielsen, Automating the assignment of submitted manuscripts to reviewers, in: Research and Development in Information Retrieval, 1992, pp. 233–244.
[18] J.I. Maletic, A. Marcus, Using latent semantic analysis to identify similarities in source code to support program understanding, in: Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000), 2000, pp. 46–53.
[19] S. Kawaguchi, P.K. Garg, M. Matsushita, K. Inoue, Mudablue: An automatic categorization system for open source repositories, in: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), 2004, pp. 184–193.
[20] A. Marcus, J.I. Maletic, Identification of high-level concept clones in source code, in: Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001), 2001, pp. 107–114.
[21] A. De Lucia, F. Fasano, R. Oliveto, G. Tortora, Enhancing an artefact management system with traceability recovery features, in: Proceedings of the 20th IEEE International Conference on Software Maintainance (ICSM 2004), 2004, pp. 306–315.
[22] A. Marcus, D. Poshyvanyk, The conceptual cohesion of classes, in: Proceedings of the International Conference on Software Maintenance (ICSM 2005), IEEE Computer Society Press, Los Alamitos, CA, 2005, pp. 133–142.
[23] T. Landauer, S. Dumais, The latent semantic analysis theory of acquisition, induction, and representation of knowledge, Psychological Review 104/2 (1991) 211–240.
[24] T.J. Biggerstaff, B.G. Mittbander, D. Webster, The concept assignment problem in program understanding, in: Proceedings of the 15th International Conference on Software Engineering (ICSE 1993), IEEE Computer, 1993.
[25] L. Wittgenstein, Philosophische Untersuchungen, Blackwell, Oxford, 1953.
[26] A. Kuhn, O. Greevy, T. Gîrba, Applying semantic analysis to feature execution traces, in: Proceedings of Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005), 2005, pp. 48–53.
[27] M.F. Porter, An algorithm for suffix stripping, Program 14 (3) (1980) 130–137.
[28] S.T. Dumais, Improving the retrieval of information from external sources, Behavior Research Methods, Instruments and Computers 23 (1991) 229–236.
[29] J. Bertin, Sémiologie graphique, Les Re-impressions des Editions de l'Ecole des Hautes Etudes En Sciences Sociales, 1973.
[30] A.K. Jain, M.N. Murty, P.J. Flynn, Data clustering: A review, ACM Computing Surveys 31 (3) (1999) 264–323.
[31] E.R. Tufte, The Visual Display of Quantitative Information, 2nd ed., Graphics Press, 2001.
[32] A. Kuhn, Semantic clustering: Making use of linguistic information to reveal concepts in source code, Diploma Thesis, University of Bern, March 2006.
[33] P. Nakov, A. Popova, P. Mateev, Weight functions impact on lsa performance, in: Proceedings of the EuroConference Recent Advances in Natural Language Processing (RANLP 2001), 2001, pp. 187–193.
[34] P. Nakov, Latent semantic analysis for german literature investigation, in: Proceedings of the International Conference, 7th Fuzzy Days on Computational Intelligence, Theory and Applications, Springer Verlag, London, UK, 2001, pp. 834–841.
[35] B. Caprile, P. Tonella, Nomen est omen: Analyzing the language of function identifiers, in: Proceedings of the 6th Working Conference on Reverse Engineering (WCRE 1999), IEEE Computer Society Press, 1999, pp. 112–122.
[36] N. Anquetil, T. Lethbridge, Extracting concepts from file names; a new file clustering criterion, in: International Conference on Software Engineering (ICSE 1998), 1998, pp. 84–93.
[37] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, Reading, MA, 1999.
[38] W. Frakes, B. Nejmeh, Software reuse through information retrieval, SIGIR Forum 21 (1-2) (1987) 30–36.
[39] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, Information retrieval models for recovering traceability links between code and documentation, in: Proceedings of the International Conference on Software Maintenance (ICSM 2000), 2000, pp. 40–49.
[40] A. Marcus, J. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, in: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), 2003, pp. 125–135.
[41] M. Lungu, M. Lanza, T. Gîrba, Package patterns for visual architecture recovery, in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), IEEE Computer Society Press, Los Alamitos, CA, 2006, pp. 183–192.
[42] M. Lormans, A. van Deursen, Can lsi help reconstructing requirements traceability in design and test? in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), IEEE Computer Society, 2006.
[43] G.A. Di Lucca, M. Di Penta, S. Gradara, An approach to classify software maintenance requests, in: Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM 2002), 2002, pp. 93–102.
[44] J. Huffman-Hayes, A. Dekhtyar, S.K. Sundaram, Advancing candidate link generation for requirements tracing: The study of methods, IEEE Transactions on Software Engineering 32 (1) (2006) 4–19.
[45] D. Čubranić, G. Murphy, Hipikat: Recommending pertinent software development artifacts, in: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), ACM Press, New York, NY, 2003, pp. 408–418.