Constructing a Reading Guide for Software Product Audits

Remco C. de Boer and Hans van Vliet
Department of Computer Science
Vrije Universiteit, Amsterdam, the Netherlands
{remco, hans}@cs.vu.nl

Abstract

Architectural knowledge is reflected in various artifacts of a software product. In the case of a software product audit this architectural knowledge needs to be uncovered and its effects assessed, in order to evaluate the quality of the software product. A particular problem is to find and comprehend the architectural knowledge that resides in the software product documentation. The amount of documents, and the differences in for instance target audience and level of abstraction, make it a difficult job for the auditors to find their way through the documentation. This paper discusses how the use of a technique called Latent Semantic Analysis can guide the auditors through the documentation to the architectural knowledge they need. Using Latent Semantic Analysis, we effectively construct a reading guide for software product audits.

1. Introduction

The architectural design of a software product and the architectural design decisions taken play a key role in software product audits. Architectural design decisions and their rationale provide for instance insight into the trade-offs that were considered, the forces that influenced the decisions, and the constraints that were in place. The architectural design that is the result of these decisions allows for comprehension of such matters as the structure of the software product, its interactions with external systems, and the enterprise environment in which the software product is to be deployed. Following a recent trend in software architecture research (e.g. [5, 12, 13, 20]) we refer to the collection of architectural design decisions and the resulting architectural design as 'architectural knowledge'.

For a given software product there is no single source that contains or provides all relevant architectural knowledge. Instead, architectural knowledge is reflected in various artifacts such as source code, data models, and documentation. A complicating factor in distilling relevant architectural knowledge from software product documentation is the fact that there are often many different documents. Each of these documents is tailored to specific stakeholders and different documents can therefore reflect architectural knowledge at different levels of abstraction. A high-level project management summary, for instance, will reflect architectural design decisions and their effects differently than a document describing detailed technical design.

The ISO/IEC 14598-1 international standard [10] defines a software product as 'the set of computer programs, procedures, and possibly associated documentation and data'. Quality is defined as 'the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs', while quality evaluation is 'a systematic examination of the extent to which an entity is capable of fulfilling specified requirements'. Consequently, when we refer in this paper to a software product quality audit - i.e. an audit in which the quality of a software product is evaluated - we refer to 'the systematic examination of the extent to which a set of computer programs, procedures, and possibly associated documentation and data are capable of fulfilling specified requirements'.

We have conducted a case study at a company that has broad experience in performing software product audits. This company provides independent quality audits of software products. Its customers range from large private companies to governmental institutions. In this case study we have investigated the use of architectural knowledge in software product audits. To this end we observed an audit that was being conducted for one of the company's customers. We attended and observed the audit team meetings and had discussions with the audit team members on their use of architectural knowledge in the audit. In addition, we held more general interviews on this topic with five employees who had been involved in various audits, two of whom were also directly involved in the observed audit. The interviewed employees possess different levels of experience and have different focal points when conducting an audit. The problem sketched above corresponds to a problem that is perceived by all auditors as being difficult to deal with.

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

In short, the auditors need a reading guide that guides them through the documentation.

In this paper, based on the case study we performed, we outline the problem of discovering architectural knowledge in software product documentation and present a technique that can be used to alleviate this problem. This technique, Latent Semantic Analysis, uses a mathematical technique called Singular Value Decomposition to discover the semantic structure underlying a set of documents. We employ this latent semantic structure to guide the auditors through the documentation to the architectural knowledge needed.

The remainder of this paper is organized as follows. The next section discusses the use of architectural knowledge in software product audits based on our observations in the case study we conducted. Section 3 presents Latent Semantic Analysis (LSA) and its mathematical background. Section 4 discusses the application of LSA to a set of documents that contain software product documentation and shows how we can employ the semantic structure uncovered by LSA to guide the auditor to relevant architectural knowledge. Section 5 contains a discussion on related work regarding the application of LSA to similar problems as well as related work in the area of research into architectural knowledge. Section 6 outlines research areas that are still open for further study, and Section 7 contains concluding remarks on this paper.

2. Architectural Knowledge in a Software Product Audit

In a software product audit, two types of architectural knowledge can be distinguished. On the one hand there is architectural knowledge pertaining to the *current state* of the software product; this knowledge reflects the architectural decisions *taken*. On the other hand there is architectural knowledge pertaining to the *desired state* of the software product; this knowledge reflects the architectural decisions *demanded* (or expected). It is the auditor's job to compare the current state with the desired state.

In order to perform a comparison of current state and desired state, the auditor has to have a firm grasp on both types of architectural knowledge. A common method to structure the architectural knowledge of the desired state is to define a number of review criteria. These criteria can be phrased as (architectural) decisions, and are a combination of the wishes of the customer and the expertise of the auditor. An example of such a criterion might be 'All errors in the software are written to a log. Each log entry contains enough information to determine the cause of the error.'. A software product audit consists of a comparison of these review criteria against the current state of the software product.

The 'current state' architectural knowledge of the software product is reflected in different artifacts, in particular in source code and the accompanying documentation. Some architectural knowledge, for instance alternative solutions that were considered but have been rejected, might not be explicitly captured in these artifacts at all. This architectural knowledge lives tacitly in the heads of its originators. Particular methods that are used to distill the architectural knowledge needed from these three sources - source code, documentation, and people - are:

- scenario analysis,
- interviews,
- source code analysis, and
- document inspection.

Both interviews and scenario analysis are techniques to elicit architectural knowledge from people's minds, and consequently require extensive interaction with the software product supplier. Source code analysis and document inspection, however, are performed using only the artifacts that have been delivered as part of the software product. In terms of availability of resources, the latter two are hence to be preferred. In the remainder of this paper we will focus on document inspection in particular. A typical first use of the architectural knowledge reflected in the documentation is for auditors to familiarize themselves with a software product. Once a certain level of comprehension has been attained, the documents are used as a source of evidence for findings regarding the software product quality.

While document inspection is an important method in a software product audit, it can also be a difficult method to use. The difficulty of performing document inspection lies in the sheer amount of documentation that accompanies most software products. Auditors are swamped with documentation, and there is no single document that contains all architectural knowledge needed. Moreover, a 'reading guide', which tells the auditors which information can be found where, is usually not available up front. Auditors need to fall back on interviews, a resource-intensive technique, to gain an initial impression of the organization of architectural knowledge in the documentation.

In general, from the interviews held we learned that auditors have three major questions regarding software product documentation and the architectural knowledge contained in it. These three questions are:

1. Where should I start reading?
2. Which documents should I consult for more information on a particular architectural topic?
3. How should I progress reading? In other words, what is a useful 'route' through the documentation to gain a sufficient level of architectural knowledge?

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

From the above it should be clear that the auditors who perform a software product audit would greatly benefit from tools and techniques that can direct them to relevant architectural knowledge. We refer to the goal of such tools and techniques as 'Architectural Knowledge Discovery' [3]. A core capability of Architectural Knowledge Discovery is the ability to grasp the semantic structure, or meaning, of the software product documentation. Employing this structure transforms the set of individual texts into a collection that contains architectural knowledge elements and the intrinsic relations between them. A technique that can be deployed to support the discovery of directions to relevant architectural knowledge is Latent Semantic Analysis.

3. Latent Semantic Analysis

A method that can be used to capture the meaning of a collection of documents is the construction of a vector-space model. Vector-space models are based on the assumption that the meaning of a document can be derived from the terms that are used in that document. In a vector-space model, a document d is represented as a vector of terms $d = (t_1, t_2, ..., t_n)$, with t_i $(i = 1, 2, ..., n)$ being the number of occurrences of term i in document d [16].

Figure 1 depicts a matrix based on the vector-space model constructed for three texts that were taken from the documentation of a software product. The three texts used are representative selections from a use case definition (UC), a service specification (SVC), and an architecture description (ARCH). Together, the three document vectors corresponding to these three texts contain approximately 90 distinct terms. This so-called term-document frequency matrix represents the number of occurrences of each of these terms in each of the three documents. The original document vectors are hence extended with terms that did not occur in the document itself, but do occur in one of the other texts. In these extended document vectors t_i is set to 0 if term i does not occur in the document. The cutout shows the exact number of occurrences of six terms in the respective texts. For reasons of non-disclosure, the terms 'domain entity', 'use case', and 'business object' have been substituted for the product-specific terminology.

Although the vector-space model in Fig. 1 captures some of the semantics of the three texts, parts of the underlying semantic relationships are not represented very well. Based on Fig. 1 we can for instance only conclude that in theory neither the use case definition nor the service specification has anything to do with the term 'SOA' (an abbreviation for 'Service Oriented Architecture'). In practice, however, we would expect at least some relevance of the term 'SOA' in the context of a *service* specification. Latent Semantic Analysis allows us to exploit these underlying, or latent, semantic relationships.

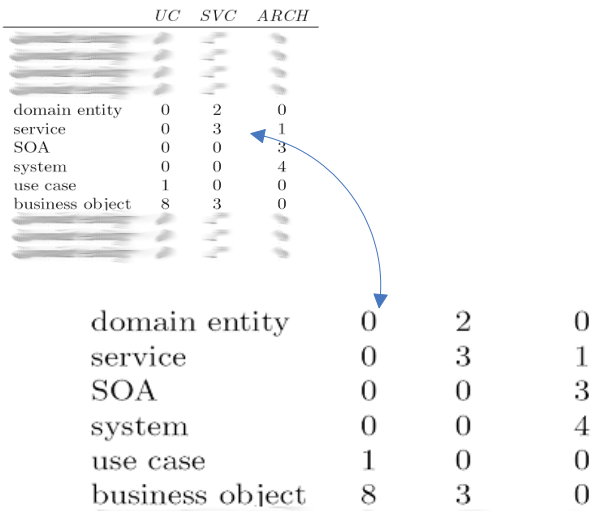


Figure 1. Term-document frequency matrix based on the vector-space model for three software product documentation excerpts.

Latent Semantic Analysis (LSA) relies on a mathematical technique called Singular Value Decomposition (SVD). SVD decomposes a rectangular m-by-n matrix A in the product of three other matrices: $A = U\Sigma V^T$. The matrix Σ is a r-by-r diagonal matrix, in which the diagonal entries $(\sigma_1, \sigma_2, ..., \sigma_r)$ are singular values and r is the rank of A. As explained in [6], SVD is closely related to standard eigenvalue-eigenvector decomposition of a square symmetric matrix. In fact, U is the matrix of eigenvectors of the square symmetric matrix AA^T, while V is the matrix of eigenvectors of A^TA. Σ^2 is the matrix of eigenvalues for both AA^T and A^TA. The interested reader can find more technical details on SVD in advanced linear algebra literature such as [7].

Since SVD can be applied to any rectangular matrix, it can also be used to decompose a term-document frequency matrix such as the one depicted in Fig. 1. The matrices U and V then contain vectors that specify the locations of the terms and documents in a term-document space, respectively. The r orthogonal dimensions in this space can be interpreted as representations of r abstract concepts (cf. [15]). The left-singular and right-singular vectors u_i and v_j indicate how much of each of these abstract concepts is present in term i and document j.

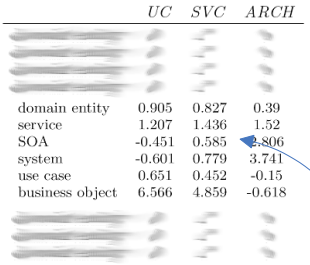
As outlined above, the original matrix A can be reconstructed by calculating the product of $U\Sigma V^T$. Instead of a reconstruction, a rank-k *approximation* of A can be calculated by setting all but the highest k singular values in Σ to 0. This approximation, A_k, is the closest rank-k approximation to A [1]. Calculating A_k for a term-document space,

such as the one depicted in Fig. 1, results in the closest k-dimensional approximation to the original term-document space [16]. In other words, by using SVD it is possible to reduce the number of dimensions in a term-document space. It is exactly this capability of SVD that is employed by LSA.

By using only k dimensions to reconstruct a term-document space, LSA no longer recalculates the exact number of occurrences of terms in documents. Instead, LSA estimates the number of occurrences based on the dimensions that have been retained. The result is that terms that originally did not appear in a document might now be estimated to do appear, and that other words that did appear in a document might now have a lower estimated frequency [15]. This is the way in which LSA infers the latent semantic structure underlying the term-document space, and the way in which the deficiencies in the semantics captured in a vector-space model are overcome.

In the reduced dimensional reconstruction of the term-document space, the meaning of individual words is inferred from the context in which they occur. This means that LSA largely avoids problems of synonymy, for instance introduced because two different authors of documentation for the same software product use two different terms to denote the same concept. One of the authors might for instance use the full product name in the documentation, while the other author prefers to use an acronym. Since the contexts in which these different terms are used will often be similar, LSA will expect the product acronym to occur with relatively high frequency in texts where the full product name is used and vice versa. However, it should probably be stressed here that we cannot expect LSA to improve the documentation in any other way than making it more accessible. LSA will happily accept wrong, superfluous, or obsolesced documentation and guide anyone interested to 'relevant' parts of that documentation. Nonetheless, for reasonably well-written documentation the latent semantic structure LSA infers can be very well exploited to guide the reader.

Figure 2 shows the result of the application of LSA to the term-document frequency matrix from Fig. 1. The cutout shows the same six terms that are shown in the cutout in Fig. 1, but this time the numbers correspond to the *estimated* term frequencies based on retaining only 2 dimensions. Upon inspection of this result, interesting patterns appear. For starters, the term SOA is now expected to be present in the service specification as well, albeit at a lower frequency than in the architecture description. This corresponds to our intuitive notion that we would expect at least some relevance of SOA to a service specification. The negative expected frequency of SOA in the use case specification is somewhat awkward to interpret mathematically, but might perhaps best be regarded as a kind of 'surprise factor'. In a sense, LSA tells us that it does not merely not



domain entity	0.905	0.827	0.39
service	1.207	1.436	1.52
SOA	-0.451	0.585	2.806
system	-0.601	0.779	3.741
use case	0.651	0.452	-0.15
business object	6.566	4.859	-0.618

Figure 2. Estimated term-document frequencies after the application of LSA to the matrix in Fig. 1.

expect the term SOA to crop up in the use case specification (estimated number of occurrences = 0); it would even be quite surprised to encounter this term there.

In general, a pattern seems to emerge in Fig. 2. If we regard the use case specification as the lowest level of abstraction text, the architecture description as the highest level, and the service definition somewhere in between, we see that low-level concepts (such as 'business object' and 'use case') have a diminishing level of association as the level of abstraction of the text increases and vice versa. This pattern stems from the semantic structure in the documents. We can employ the uncovered semantic structure to guide an auditor to the information needed.

4. Constructing a Reading Guide: A Case Study

The LSA technique introduced in Section 3 forms the basis of a detailed case study in which we examine how the semantic structure discovered by LSA can be employed to guide the auditors through the documentation. This section presents the results of this case study.

Figure 3 depicts the interactive process by which an auditor is guided through the documentation. Initially, auditors start with a set of unread documents. Although the content of these documents is still unknown, the auditors have a goal that needs to be satisfied by reading (part of) the documentation. Examples of such goals are obtaining a global understanding of the software product, investigating certain quality attributes, or locating (further) evidence for

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

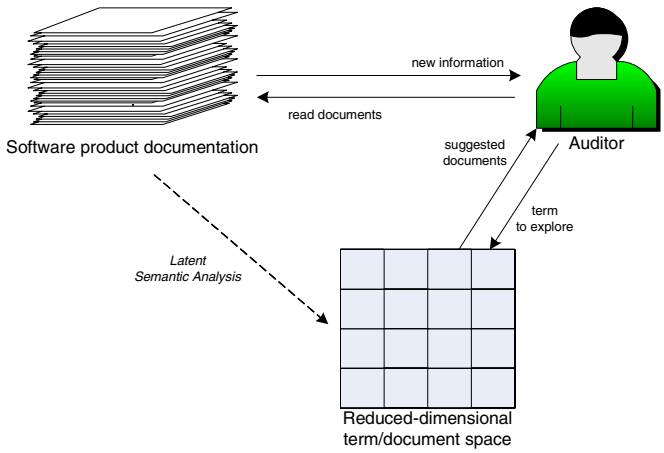


Figure 3. Schematic overview of the construction of a reading guide using the reduced-dimensional term-document space calculated by LSA.

certain findings. The reduced-dimensional term-document space, which results from LSA, can be inspected to locate documents that are highly associated with a term that corresponds to the auditor's goal. For instance - and this example will be worked out in more detail below - the term 'architecture' could be used to find documents that provide high-level information about the software product. From reading the suggested documents, an auditor learns new information including new - potentially product-specific - terms that can be used to locate documents that provide more detail on the new terms. In short, reading guidance consists of an iterative process of selecting and reading documentation, in which the auditor can use the architectural knowledge gained from reading suggested documents to steer the selection of new documents.

We applied LSA to a total of 80 documents that were subject to the audit that has been described earlier. The term-document frequency matrix that was constructed for these documents contained a total of 3290 terms found in the 80 documents. These 3290 terms did not contain very common words ('stopwords') such as 'a', 'the', or 'is'. It is common practice to disregard these stopwords, since they tend to be evenly spread over all documents and hence do not bear any distinctive meaning. The length of the document vectors that make up the term-document frequency matrix had been normalized before LSA was applied. This normalization reduces the effect of document size (i.e. the number of terms in the document); without normalization, longer documents tend to be favored over shorter documents when a document selection is made.

Using the technique described in Section 2, we calculated a 5-dimensional approximation of the constructed term-document frequency matrix. The selection of the number of dimensions to retain is an empirical issue [15], although some heuristics exist [2]. The rank-5 approximation chosen here requires a 49% change relative to the original term-document frequency matrix. Although this might appear as a rather large change, the results obtained with this approximation suit our needs; they can be effectively used to construct a reading guide.

The case study presented here reconstructs the early phase of the software product audit, in which the auditors need to attain a global understanding of the software product in order to further assess its quality. As in the previous section, for reasons of non-disclosure the results presented here have been anonymized.

In general, when auditors commence a software product audit they want to gain an initial, high-level understanding of the software product. This global understanding is necessary to successfully perform the audit, since it is a prerequisite for subsequent audit activities. For instance, in scenario analyses the supplier of the software product is asked how the product reacts to certain change scenarios or failure scenarios. In order to judge the answer the supplier provides, an auditor needs to have a thorough understanding of the software product. Otherwise, the supplier might provide an answer that is incomplete or inconsistent with the real state of the product, without this being noticed.

Auditors who want to attain overall comprehension of the software product can be guided through the documentation using the semantic structure discovered by LSA. A route that is preferred by all auditors we interviewed is to start with high-level, global information and gradually descend to texts that contain more detailed and fine-grained information. A single term that can be expected to cover the high-level information about the software product well is the term 'architecture'.

We can inspect the reduced 5-dimensional approximation of the original term-document frequency matrix that LSA has calculated to find the documents that best match the term 'architecture'. In order to do so, it suffices to rank the documents by their respective values in the row that coincides with the term 'architecture' (the 'architecture' term vector). Documents that have a high value in the 'architecture' term vector correspond to the documents in which LSA expects the highest number of occurrences of the term 'architecture'. Recall that the highest-ranking documents do not necessarily include the literal term 'architecture', but that LSA inferred that it would be likely to encounter the term 'architecture' in these documents; they are semantically close to the meaning of 'architecture'. In other words, these documents talk *about* architecture, perhaps without mentioning the word 'architecture' itself.

The list in Table 1 shows the 10 highest ranked documents for the term 'architecture', together with the actual

Table 1. Top-10 documents that match the term 'architecture', with the number of occurrences of 'architecture' in the document.

Rank	*Document ID*	*# 'architecture'*
1	79	44
2	39	1
3	44	3
4	41	2
5	78	10
6	46	0
7	45	1
8	42	1
9	40	2
10	49	0

number of occurrences of 'architecture' in these documents. Given this list, an auditor can simply start reading top-down, in this case starting with document 79. However, some of these documents are fairly large while others are rather small. In fact, documents 46 and 45 both consist of only 2 pages. If an understanding of the software product can be attained by either reading a (very) small document or by ploughing through a large number of pages, the former is obviously preferred by the auditors. Table 2 lists the same top-10 documents for 'architecture' also shown in Table 1. In this table, however, the documents have been categorized according to their size. The size categories have been defined as: very small (< 5 pages), small (< 10 pages), medium (< 30 pages) and large (≥ 30 pages). The rank according to Table 1 is given in between brackets, to illustrate the differences.

Table 2. Top-10 documents that match the term 'architecture', grouped by size.

	Rank		*Doc. ID*	*# pages*
Very small	1	(6)	46	2
	2	(7)	45	2
Small	3	(5)	78	6
	4	(10)	49	9
Medium	5	(1)	79	24
	6	(2)	39	13
	7	(3)	44	21
Large	8	(4)	41	30
	9	(8)	42	48
	10	(9)	40	31

Table 2 shows that, given a preference for smaller documents, an auditor looking for information about the architecture of the software product should first read document 46. Note that this document does not contain the term 'architecture' at all (see Table 1). Nevertheless, upon inspection this document indeed contains high-level 'architectural' information.

From document 46, the auditor learns that the software product consists of three high-level components, which we will call X, Y, and Z. Furthermore, the document identifies two external systems that interact with the software product as well as an organizational unit that will handle certain types of operational problems that might occur. Finally, the document contains a list of intended uses of the software product.

Now that the auditor knows a little more about the software product, the next document has to be selected. Since the auditor still has not read all 'architecture' documents, there are in principle two options: either remain with the 'architecture' documents and select a document from that list (e.g. document 45) or use the architectural knowledge obtained to delve into a particular topic.

Good candidates for further exploration of the documentation are the components X, Y, and Z. Since these components conceptually divide the software product in three distinct parts, auditors might want to examine each of these parts to further their global understanding.

In its current form, the selection of the right terms for exploration is a matter of experience. It is from experience that the auditor knows that the term 'architecture' is likely to be related to high-level software product documentation. It is from experience that the auditor suspects that the three components are good candidates for further exploration.

In order to assess the deviation of each of the components from the meaning of the term 'architecture' a calculation can be performed of the similarity between the terms 'architecture' and 'componentX', 'componentY', and 'componentZ' respectively. This enables us to identify how much the texts for which LSA infers a high association with each of the components deviate from the text in document 46, which closely resembled the meaning of the term 'architecture'. Given the fact that the auditors want to gradually progress through the documentation, the degree of deviation is an indication of the route that should be followed through the documents.

A common measure of similarity between terms (or 'term-term similarity') is the cosine of the angle between the two term vectors [2]. Let t_i and t_j be the term vectors for terms 'i' and 'j' respectively, i.e. the rows from A_k that correspond to the terms 'i' and 'j'. Then the cosine of the angle θ between these term vectors is $\cos\theta = \frac{t_i \cdot t_j}{||t_i||||t_j||}$.

Table 3 shows the similarity of each of the terms 'componentX', 'componentY', and 'componentZ' with the term 'architecture', calculated as the cosine between their respective term vectors. It becomes clear from these three values that 'componentX' is semantically closest to 'architecture'

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

followed by 'componentZ', and that 'componentY' is the least similar to 'architecture'.

An interesting observation is that the relations between the three components are not readily apparent from the text in the document itself, nor from the names given to the components. Although the document does contain a picture that seems to suggest a layered ordering of the components, the text in document 46 does not mention or reflect such a layered approach at all. Here, by using LSA we have truly discovered architectural knowledge that the auditor could not have distilled from reading document 46 alone. This discovered knowledge can be used to further explore the documentation.

Based on the similarity of 'architecture' and each of the three components, a logical next step to read the documentation seems to first read the top-ranking documents for 'componentX', then for 'componentZ', and finally for 'componentY'. By following this route, the semantical distance between the document just read and the newly selected documents increases gradually.

Table 3. Cosine term-term similarity of 'architecture' and the high-level components

componentX	*componentY*	*componentZ*
0.9814	0.4096	0.7900

Analogous to the selection of the top-10 documents for the term 'architecture', we selected the top-10 documents for each of the terms 'componentX', 'componentY', and 'componentZ'. For each of the components, we analyzed the top-10 documents found. The results of this analysis show an interesting pattern in the selection of documents.

Due to its close semantical resemblance of 'architecture', shown in Table 3, the top-10 documents that were found for the term 'componentX' are in fact the same 10 documents that were found for 'architecture'. The only difference is a small change in the ranking of the documents. The top-10 documents found for the term 'componentZ' (the next closest term to 'architecture') comprises a mix of service specifications and architectural design descriptions, with a clear focus on service specifications (the first four documents in the list are service specifications). The top-10 documents found for 'componentY' are all use case definitions.

The route through the documentation found by analyzing the result of LSA suggests that, using 'architecture' as a starting point, the auditors should first read the architecture descriptions and similar high-level documentation, then proceed with service specifications, and finally read the use case definitions. Although LSA does not in and of itself know of the distinction between high-level documents (i.e. architecture descriptions) and low-level documents (i.e. use case definitions), the documents that it suggests to read are grouped along this axis. Moreover, from interviews with the auditors we learned that this is indeed a route they prefer to follow to familiarize themselves with a software product. This proves the value of the use of LSA as an architectural knowledge discovery technique to construct a reading guide for a software product audit.

5. Related Work

The application of Latent Semantic Analysis to architectural knowledge discovery discussed in this paper bears some relation to other work, both within and outside of the Software Engineering research domain. The origin of LSA lies in information retrieval. LSA was presented in 1990 by Deerwester et al. as 'a new method for automatic indexing and retrieval' of documents [6]. Later research also focused on the psycholinguistic significance of LSA. Landauer and Dumais, for instance, use LSA to simulate the acquisition of vocabulary from text, and present LSA as a theory of acquisition, induction, and representation of knowledge [14].

Over the years, LSA has seen various application domains, including Software Engineering. For instance, Maletic et al. applied LSA to source code of software components in order to support program comprehension [17, 18]. Another approach is taken by Hayes et al., who use LSA to support the construction of requirements traceability matrices [8].

Our approach adds a new item to the list of LSA applications in Software Engineering. Although construction of a reading guide for software product documentation also contributes to better program understanding, our approach differs from the use of LSA by Maletic et al., who apply LSA to source code artifacts. The effect of a reading guide is also not limited to better product comprehension, but further supports the auditors in locating evidence for their findings. Since architectural knowledge can be reflected differently in source code and documentation, some of the evidence and knowledge that can be found in the documentation might not be available from the software product's source code.

6. Future Work

The work presented in this paper gives rise to a number of issues that warrant further research. An overall issue that remains to be investigated is the scalability of our approach. LSA proved to be feasible for a corpus of 80 documents, but in practice software product documentation might comprise many more documents. Document sets of several hundreds of documents are not uncommon. Also, our current use of LSA concerns a final set of documents. We may also envision using LSA in a forward-engineering sense, to judge

the quality of the evolving (architectural or otherwise) documentation of a system, and giving guidelines as to where the documentation needs attention.

Besides these global issues, we have identified four main areas that are to be further explored in the present context: enhancement of the workflow, the use of background knowledge, quantitative evaluation, and user interaction. This section describes each of these areas in more detail.

6.1. Workflow Enhancement

The 'workflow' presented in this paper, i.e. the selection of terms to explore the documentation, is still rather ad hoc and depends heavily on the auditor's experience. One could wonder whether the same result would have been obtained had the process been started with another 'high-level' term, such as the name of the software product instead of the generic term 'architecture'.

As a matter of fact, using the name of the product, or the high-level term 'system' instead of 'architecture', would have yielded a different result. The documents that are suggested for these terms resemble the documents that were suggested for 'componentZ', i.e. with an emphasis on service specifications. Document 46 would not have been suggested for any of these terms. Depending on one's opinion this may or may not come as a surprise. Some might argue that 'system' indeed carries more of a notion of implementation than 'architecture'. It does show, however, the importance of the selection of the (initial) terms to explore. It also shows that the auditor would benefit from assistance in this selection instead of having to rely on experience alone. We would like to capture this kind of experience to enhance the workflow and aid the auditor in selecting new terms to explore.

We believe that we can capture relevant experience and enhance the workflow by introducing a feedback loop. By keeping track of terms that worked well in earlier projects, the auditor can be presented with suggestions as to which terms to explore in a new project. This helps the auditor to get the project started (e.g. by suggesting initial terms such as 'architecture'), but can also circumvent potential dead-ends in the exploration by explicitly ignoring terms that are known to have led to dead-ends in previous audits. Such suggestions could perhaps also be mined from the documentation itself, using techniques such as frequency profiling to locate uncommon words (with respect to a standard corpus) that are likely to be part of a domain-specific vocabulary. Sawyer et al. report successful application of frequency profiling in extraction of domain terms from requirements engineering documents [19].

While a feedback loop could relatively easily handle common generic terms such as 'architecture', additional research is needed in particular to determine how to cope with product-specific terminology such as 'componentX'. The reasons that auditors choose certain (product-specific) terms for further exploration need to be analyzed and translated to more generic 'heuristics' and best practices that are applicable to other projects as well. An example heuristic might be that, given the auditor's goal of overall product comprehension, terms that signify components are better candidates for further exploration than terms that signify intended uses. This heuristic can change when the auditor's goal changes. If the auditor is looking for the satisfaction of certain requirements, intended uses might be preferred over components.

6.2. Background Knowledge Incorporation

The 'queries' that are used in this paper to explore the software product documentation are logical from an auditor's point of view. The auditor starts with a high-level exploration of the software product's architecture, gradually zooming in to reveal more detailed architectural knowledge. Through Latent Semantic Analysis, documents with diminishing levels of abstraction are identified: from architecture descriptions at the highest level through service definitions to use case specifications at the lowest level. However, this analysis sequence still requires extensive human interpretation. As remarked earlier, LSA itself has no notion of 'high-level' or 'low-level' documentation, nor of any other domain-specific knowledge.

To further enhance the support for auditors reading the software product documentation we intend to investigate the incorporation of relatively static background knowledge in the automated analysis of the documentation. The words 'relatively static' signify domain knowledge that does not change for each audit. Apart from often used classifications such as high-level vs. low-level documentation, examples of such background knowledge are:

- generic models, such as quality models (e.g. [11]) and process models (e.g. [9]);
- ontologies, for instance of architectural patterns (e.g. [4]) and their known relations with for example quality attributes;
- 'heuristics', such as an auditor's general preference for smaller documents (see also Section 4).

Background knowledge can be incorporated in construction of a reading guide by using it in the selection of suggested documents to read. Models and ontologies can for instance be used to broaden the scope when exploring a certain term; they can be used to formulate rules of the form 'if auditors are interested in X (e.g. 'maintainability') they are (probably) also interested in Y (e.g. 'changeability', see also [11])'. Heuristics can for example be applied to change

the suggested order in which the documents should be read, as demonstrated in Section 4.

Note that there is also some overlap of background knowledge incorporation with the planned workflow enhancements described in Section 6.1. The heuristics (or best practices) described in that section can in fact be regarded as background knowledge. The translation of product-specific terminology to generic terms used in these heuristics could very well be based on an ontology structure.

Background knowledge can hence be employed at two levels: to suggest terms to explore, steering the workflow; and to suggest documents to read, steering the analysis. Both affect the outcome of the process, the reading guide.

6.3. Quantitative Evaluation

This paper shows that the application of LSA to the construction of a reading guide for software product audits delivers results that support the auditors in finding a route through the documentation. However, we do not have a quantitative measure of 'goodness' of the reading guide.

We intend to further research the quantitative evaluation of architectural knowledge discovery, and in particular evaluation and validation of the application of LSA discussed in this paper. We are currently already investigating the use of the so-called repertory grid technique to elicit the mental model an auditor constructs and/or uses in a software product audit. A mathematical comparison of the distances between documents and/or concepts as they are perceived by the auditors with the distances that are calculated by LSA then leads to a quantitative measure of 'goodness'. Preliminary experience with this technique leads us to believe that the repertory grid technique can be used to validate the result of architectural knowledge discovery.

Having defined a quantitative evaluation measure, we can closely monitor whether adjustment of our method improves the result. We could, for instance, assess the effect of the incorporation of background knowledge in the analysis of the documentation. Such an assessment consists of a comparison of the distances perceived by the auditors with the distances before and after incorporating background knowledge. If the result of the analysis corresponds more to the auditor's mental model when background knowledge is taken into account, this means that using this background knowledge is indeed an improvement. A quantified comparison of the auditor's mental model with the result of LSA could also provide guidance to selection of the right number of reduced dimensions, since the optimal number of dimensions yields the best match of the LSA result with the auditor's perception. Finally, the effect of using other techniques instead of, or together with, LSA could be easily judged analogous to the assessment of the incorporation of background knowledge. Techniques that could further improve architectural knowledge discovery results include techniques that, complementary to LSA, exploit certain more structured properties of the documentation. If, for example, a set of documents is structured according to a particular template – which is not uncommon – knowledge of this template could be used to guide the reader to particular parts of the documentation.

6.4. User Interaction and Tool Support

A final area in which further research is needed is the area of user interaction. The results presented in this paper all show direct operations on the reduced-rank approximation of the original term-document frequency matrix. This matrix is arguably not the best form of presentation for the end users, i.e. the auditors.

In order to be useful and used in an auditor's everyday practice, the techniques discussed in this paper should be implemented in an interactive environment that abstracts away from the underlying estimated term frequencies. This environment should provide intuitive support for the workflow discussed in Section 6.1.

A particular area that requires further research is visualization of the reduced-dimensional term-document space. A desirable visualization supports both locating terms to explore and locating documents to read. Ideally, this would be presented to the auditors as a space through which they could navigate in search of the architectural knowledge they need. In this space, distances between terms and/or documents have actual meaning (cf. the three terms in Table 3). Such a visualization requires a projection of the reduced-dimensional term-document space to two - or at most three - dimensions. Through such a visualization, auditors can obtain quick visual clues as to which documents are closely related and how to proceed reading the document set.

7. Conclusion

Document inspection is a method used in software product audits to distill architectural knowledge from the software product documentation. Unfortunately, document inspection is often hard to perform. Auditors are in need of a reading guide that tells them where to start reading, how to progress reading, and which documents to consult for more detail on a particular topic.

We have demonstrated how auditors can be guided through the documentation with a case study in which we reconstructed the early phase of a software product. In this phase, the auditor has not read any documents yet and needs to attain a certain level of understanding of the software product.

To construct a reading guide, we have employed the semantic structure discovered by Latent Semantic Analysis.

This semantic structure is used as the basis for an interactive process in which auditors indicate terms that they want to explore and are subsequently given reading suggestions for documents containing information about these terms. The knowledge obtained from the suggested documents can give rise to new terms to explore, and the discovered semantic structure can be used to determine the order in which the terms - and corresponding documents - should be explored.

We have identified four areas of future work: workflow enhancement, use of background knowledge, quantitative evaluation, and user interaction. We intend to direct research efforts toward each of these areas in order to further improve the work presented in this paper.

Acknowledgement

This research has been partially sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. We would like to thank the anonymous reviewers for their insightful comments.

References

[1] M. Berry, S. Dumais, and G. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report CS-94-270, December 1994.

[2] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[3] R. C. d. Boer. Architectural knowledge discovery: Why and how? In *Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, Torino, Italy, 2006.

[4] G. Booch. Handbook of software architecture, http://www.booch.com/architecture/.

[5] J. Bosch. Software architecture: The next step. In F. Oquendo, B. Warboys, and R. Morrison, editors, *Software Architecture: First European Workshop (EWSA)*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199, St. Andrews, UK, 2004. Springer-Verlag GmbH.

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

[7] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[8] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Improving after-the-fact tracing and mapping: Supporting software quality predictions. *IEEE Software*, 22(6):30–37, 2005.

[9] ISO/IEC. Information technology - software product evaluation - part 5: Process for evaluators. Technical Report ISO/IEC 14598-5, 1998.

[10] ISO/IEC. Information technology - software product evaluation - part 1: General overview. Technical Report ISO/IEC 14598-1, 1999.

[11] ISO/IEC. Software engineering - product quality - part 1: Quality model. Technical Report ISO/IEC 9126-1, 2001.

[12] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, Pittsburgh, Pennsylvania, USA, 2005.

[13] P. Kruchten, P. Lago, and H. v. Vliet. Building up and reasoning about architectural knowledge. In *2nd International Conference on the Quality of Software Architectures (QoSA)*, 2006.

[14] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.

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

[16] T. A. Letsche and M. W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Sciences*, 100(1-4):105–137, 1997.

[17] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2000.

[18] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *14th IEEE international conference on Automated Software Engineering (ASE)*, 1999.

[19] P. Sawyer, P. Rayson, and K. Cosh. Shallow knowledge as an aid to deep understanding in early phase requirements engineering. *IEEE Transactions on Software Engineering*, 31(11), 2005.

[20] J. S. v. d. Ven, A. G. J. Jansen, J. A. G. Nijhuis, and J. Bosch. Design decisions: The bridge between rationale and architecture. In A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech, editors, *Rationale Management in Software Engineering*. Springer-Verlag, 2006.