Master Thesis

Information Retrieval Services for Conceptual Content Management: Evaluation and Systems Integration

Submitted by: Galip Gülsen - 23944
Information and Media Technologies

Supervised by:
Prof. Dr. Joachim W. Schmidt (STS)
Prof. Dr. Friedrich H. Vogt (TI5)
MSc. Sebastian Boßung (STS)

Technische Universität Hamburg-Harburg

Software Systems Group

Hamburg, September 2005

Abstract:

Content Management Systems are used to store different types of contents and retrieve them in an efficient way. They contain a persistence layer for large data, a file system and other software modules which are used to facilitate the content management system. On the other hand, a Conceptual Content Management System (CCMS) [CCM] aims to improve the meaning of content with asset modeling (concept-content model). This master thesis intends to develop and integrate a full-text search engine application into the CCMS. The full-text search engine application performs indexing and searching operations in order to manage and retrieve large data in CCMS efficiently.

Hereby, I declare that:

This master thesis, with the subject "Information Retrieval Services for Conceptual Content Management: Evaluation and Systems Integration", has been prepared by myself. All literal and content related quotations from other sources are clearly pointed out, and no other sources or aids than the declared ones have been used.

Galip Gülsen
Hamburg, September 2005

Hereby, I would like to thank Prof. Joachim W. Schmidt for finding this master thesis topic and supervising it. I also thank Prof. Dr. Friedrich H. Vogt for accepting being my co-supervisor.

Furthermore, MSc. Sebastian Boßung was very patient and helpful for answering my questions and providing programming tools in order to develop the applications. His advices directed the project progress for a better implementation. Thanks also to Dr. Hans-Werner Sehring for answering fundamental questions about the project.

CONTENTS

Figures:

Chapter 1

1. Introduction

1.1. Motivation

Nowadays information plays an important role in people's life and computer science. Its growth is inevitable because of fast development in internet, web, software and hardware technologies. Internet provides an easy structure to spread information worldwide. Web technologies support different standards (like HTML, XML, Web Services) in order to transfer information efficiently. Furthermore, hardware systems can store huge amount of data using different data structures (file systems, relational databases). Also, many software programs are available for creating information and publishing them in favour of users.

But the tremendous growth of information and information technologies give rise to many problems for developers and users. The most known are;

- How to reach required information?
- How to retrieve information in a quick and accurate way?
- Are the retrieved data relevant to user's requirements?
- Are the users satisfied from the search result?

Therefore, it is obvious that the information itself has to be considered. So, computer science has started to work on Information Retrieval Services. One solution for these problems was development of search engines. At the beginning, search engines looked for documents using linear search. In linear search a program must search for all documents in the system just looking their titles in order. This method took too much time and was not efficient.

But, the development in 'Information Retrieval' area introduced new mathematical models in order to improve search methods. The basic Information Retrieval models are Boolean model, Vector Space model and Probabilistic model. They are further explained in section x. Also, there are many variations of these models, for example Latent Semantic Indexing (LSI) (see section 2.3.1). Furthermore, in recently a new model was introduced by

Prof. Thomas Hofmann which is Probabilistic LSI (it is an interesting approach and details can be read from the paper [Hof99]). All these models aim to use and understand information better for retrieving efficiently.

The incorporation between Information Retrieval Services and software engineering enables powerful methods for managing and searching information. In analysis, design and implementation phases the Boolean model provides searching with Boolean operators. In addition, the Vector Space model introduces weighting of documents and ranking them. On the other hand, the Probabilistic model calculates probability of relevance between documents in order to return better search results.

Search engines are mostly developed for retrieving web pages in internet. Also, they are used as Information Retrieval services in different environments (like in an organization or enterprise framework). In order to adapt Information Retrieval methods into their systems, the frameworks implement their own search engine functionalities. Therefore, Content Management Systems (CMS) are developed in order to realize Information Retrieval methodology. CMS are a complete system that performs creation, content management, publishing and searching in a collaborative manner. CMS works well with ordinary content like textual documents or web pages. But, recently the type of content varied from text to multi-media (e.g. image, audio or video). Therefore, content management and implementing search engines become more challenging.

In this notion, a new model was derived namely Conceptual Content Management System [CCM] that defines a new entity modeling and aims to improve the meaning of content with closely coupled concept-content model. The details can be read in section 2.2. As a result, this paper intends to use the CCMS and add a full-text search engine in order to realize Information Retrieval services.

1.2. Problem Statement

In introduction part the importance of information and the role of content management systems in information are stated. Also, information retrieval is an important concept for using data effectively. Furthermore, the Conceptual Content Management System (CCMS) goes one step further and provide a new entity modeling.

In CCMS the entity consists of content and concept parts. This content-concept structure defines assets in the system. The main idea in CCMS is to add conceptual attributes to the content of data. This improves the comprehension of content in the system. Because, nowadays type of contents varies from textual documents to multi-media (e.g. image, audio, video). So, this makes the applications more difficult to retrieve information from the content and manage it. In such system structure, the available computational models lack from providing complete support for information retrieval, managing content, presenting documents, modifying them, etc. Therefore, CCMS provides an efficient modeling to associate the concepts with their content.

In this project, I proposed to investigate information retrieval services for CCMS and integrate a full-text search engine into the CCMS. Therefore, the underlying structure is based on the previously developed system CCMS. This system implements an asset modelling and stores the assets in its local database.

The problem is if the number of assets increase and the contents of assets are large, then the system needs an information retrieval application. This application is a full-text search engine that provides indexing and searching features in favour of CCMS. It facilitates to search data and read concept-content parts of an asset. The search engine user interface provides a user-friendly and natural application for inputting user queries and showing search results to the users.

1.3. Related Works

In this project, the full-text search engine was implemented as a part of Conceptual Content Management System (CCMS), which is implemented at Software Systems Institute (STS) by Hans-Werner Sehring. General information can be read from the papers [Seh04], [SS03] and [SS04]. Basically, the framework defines an asset modeling for content of data which is constructed as a content-concept model. CCMS provides asset language, asset definition language (ADL) compiler [ACF] and modeling tools. For example, the asset definition language (ADL) compiler creates a CCMS from a user's asset definitions. Furthermore, different client modules are generated from a CCMS for various platforms. The details for CCMS are investigated in section 2.2.

The implemented search engine application performs indexing and searching operations for CCMS. The content-concept parts of an asset will be easily searched and retrieved by the users. The search engine brings many advantages of information retrieval library, for example phrase, fuzzy, proximity searching and ranking of documents.

The second project related to this project topic is Warburg Electronic Library (WEL) project [Welib]. It is based on the "Image Index of Political Iconography". *"The Image Index gives an iconographic overview of the variety of phenomena which reflect political concepts, processes and demands and their relevance for the history of the arts"* as stated in [Welib] homepage. It is an open and dynamic application that uses the CCMS framework. WEL aims to store and access to multi-media documents using the CCMS's entity modeling features and functionalities.

1.4. Structure of the Thesis

This paper starts with the introduction of master thesis and its topic. Then, the topic and problems are explained further. Also, the related works are listed which describe some projects similar to the subject of this master thesis. Section 2 explains state-of-the-art technologies that are related to the project. The recent ideas like Conceptual Content Management System are stated. Furthermore, Information Retrieval (IR), content management systems, IR models and search engines are generally explained.

Section 3 gives information about Information Retrieval services which provide IR methods and processes for retrieving data from systems. The subsections further explain the IR libraries that are used for implementing full-text search engines, and the comparison results of selected search engine libraries are shown.

Then the design of application is described in Section 4. This section consists of the selected IR library – Lucene [LUC], definition of indexing, searching and analysis. Also, document parsers used in search engines, the modules in the application and the overall system structure of the developed search engine application are explained.

On the other hand, the implementation of search engine application is explained in detail. The implementation part contains Lucene indexing process, searching process, developed document parsers, parsing of user queries, analyser used in the search engine application, application logic of the program and user interface of the application that provides indexing of documents and searching them.

In Section 6, the evaluation of results is stated. The search engine facilities for Conceptual Content Management System are explained. Furthermore, some test cases are performed in order to understand Lucene library features.

Finally in section 7, the conclusions about the project and what can be done for a future work are stated.

Chapter 2

2. State of the Art

2.1. Content Management Systems

Content management systems (CMS) provide a complete framework for creating, organizing, managing and publishing content in a computer system. Nowadays, CMS are mostly used as a web application that provides these functionalities for web content and web pages. So, the process can be divided as follows [Rob03]:

- Creation of content (content modelling)
- Content management
- Publishing to users
- Presentation (listing, sorting, browsing)

CMS have a permanent storage for storing the content (e.g. databases or disks). They also provide client applications in order to administer the system, implement user tools for monitoring data, indexing new content and searching any documents from the storage correctly and efficiently. So, the user interfaces should be user-friendly and multi-functional for fulfilling user requirements. There are different types of CMS that specialize on specific areas which are known as:

- Web CMS: for web management
- Transactional CMS: for transactional operations in e-commerce
- Integrated CMS: for a specific organization
- Publication CMS: for online newspapers or bookstore
- Learning CMS: for e-learning systems
- Enterprise CMS: consists of various functionalities taken from other type of CMS.

There are some difficulties in CMS that should be taken into consideration. Updating the content of documents, retrieving required documents from the system or generally saying managing the data can be problematic. Because of the growth in internet, web and multi-media contents makes managing data in systems difficult. In this project, we consider the content and its concept. Many CMS can only provide little information about the application concepts related to their content (especially in multi-media content). In order to overcome this problem, a new model Conceptual Content Management System (CCMS) was presented and developed by Hans Werner Sehring which connects the content and concept pairs, and produce a new structure which is an asset. This model aims to improve the meaning of content. Next section explains CCMS in detail.

2.2. Conceptual Content Management System

As described in previous section content management systems implements a structure for storing, managing and outputting data for programmers and users. In general, in computer science the representation of content is straightforward and this approach is well implemented by data structures and databases in a persistent storage. Furthermore, using various monitoring tools and information retrieval applications (search engines) the content in content management systems can be retrieved efficiently. But as stated in the paper [SS03]; if the content is any type of data (e.g. multi-media: images, audio, video), the content management applications can provide limited functionalities for getting content, presenting and using it. So, there is a little support for retrieving conceptual information of the content.

Conceptual Content Management System (CCMS) constructs a new framework which is concept-content modelling. In CCMS, the concept and content of data are closely coupled in order to improve the meaning of content and they form the asset model which is the foundation of CCMS. The concept part explains characteristics, attributes and rules for the content. So, an asset is defined as concept-content pairs.

The content-concept model is defined by asset definition language (ADL). It has two main perspectives according to the entity modeling:

1) Expressiveness: This defines the three features that must be in entity modelling, therefore also in asset modeling. They are:
    - Characteristics of the asset
    - Relationships between as asset and other assets
    - Systematics that defines rules for assets

2) Responsiveness: The success of asset modeling is determined by being;

- Open: Openness means that users can adapt the pattern of the system according to their desires or requirements (e.g. adding new attributes, relationships or rules to the modelling).

- Dynamic: Dynamism means that the developed system can control and adapt these desires independently and automatically without any interference by a programmer.

The overview of entities and assets are shown in figure 2.1.

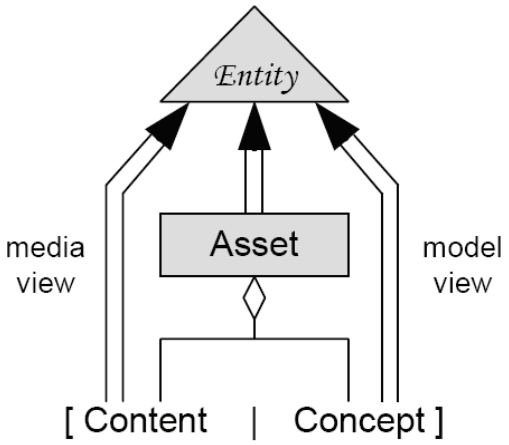


Figure 2.1: Representation of Asset Model [SS04]

As depicted in the figure, the content and concept form the asset model. The content part describes the media view (e.g. image) and the concept part describes the model view (characteristics, relationships, rules). This asset modeling was implemented in object-oriented programming; an example asset class definition is shown below.

```
class Fund {
        content
                contentIds: String
        concept
                characteristic titel: String
                characteristic datum: java.util.Calendar
                characteristic bemerkung: String
                characteristic erfassungsdatum: java.util.Calendar
                characteristic erfassungsdatum: java.util.Calendar
                characteristic typ: String
                relationship standort: Referenz
                relationship erfasser: User
                relationship verschlagwortung: Schlagwort*
                relationship kommentare: Kommentar*
                relationship masks: Mask*
}
```

The codes show that expressiveness perspectives which are 'characteristics' and 'relationships' of assets are declared. Then, these descriptions are added as attributes to the 'Fund' asset type. These attributes construct the content and concept parts of the asset. As a result, this modeling gives the possibility for end-users (programmers) easily construct new asset classes for their CCMS systems because of the asset modeling's *open* and *dynamic* features.

2.3. Information Retrieval

Information Retrieval (IR) is an interdisciplinary area that is basically described as storing data, searching documents and describing them to users. Therefore it involves many different environments. In IR data should be stored in permanent storage, for this purpose physical devices and databases are constructed. Also, the representation of files are implemented with special data structures, database design, special index structures (inverted file, graphs or trees) or continuous file types (like audio or video) etc. Besides the storage and data representation IR should provide searching among the documents and database in an efficient ways. Therefore, IR plays an important role for reaching any documents, in a quick way and resulting accurate search results.

In order to understand Information Retrieval better, the differences between IR and data retrieval are shown in figure 2.2. [Rij]. Data Retrieval is different from IR in the sense that it retrieves data from databases with exact matches.

	Data Retrieval	**Information Retrieval**
Matching	Exact match	Partial match, best match
Inference	Deduction	Induction
Model	Deterministic	Probabilistic
Classification	Monothetic	Polythetic
Query language	Artificial	Natural
Query specification	Complete	Incomplete
Items wanted	Matching	Relevant
Error response	Sensitive	Insensitive

Figure 2.2: Data Retrieval vs. Information Retrieval

If the above figure is analyzed; while matching the documents with user queries in data retrieval, the system returns an exact match or nothing (for example, using an SQL query, a row data from a table in a database is retrieved). But, in IR the system returns partial or best matches using the index. The data retrieval model is deterministic, meaning that there is no random search. A search is executed at one time and results are fetched. But, in IR there is also a probability that more relevant documents have high probability to be retrieved during searching. There is a probability that a query could not match any document from the system, too.

Also, entered query languages are different between data retrieval and IR. Data retrieval needs artificial queries like SQL, XQuery [XQu] that are more complex and must be typed correctly. On the other hand, IR systems support natural queries that are more human readable and understandable (for example, the query "Java AND Swing"). Fuzzy logic in IR services gives flexibility that although there is a syntax error in a query, the search engine can return documents similar to the query. Finally, user preferences differ between data retrieval and IR. In data retrieval the user wants to get an exact match of data specified in a query, while in IR the user wants to retrieve the best or most relevant documents that meet their preferences. Therefore, the usage areas are different between data retrieval and information retrieval systems. Search engines generally use an Information Retrieval system and its features.

When the quality of Information Retrieval Services is taken into consideration, there are two classical parameters [FB92], which are Precision and Recall (see figure 2.3). Precision is the number of relevant documents retrieved divided by number of retrieved documents. In contrast, Recall is the number of retrieved documents divided by number of relevant documents. The values reside in interval between 0 and 1. There is a trade-off between these values, for example if the Precision increases then the Recall decrease and vice versa.

$$\text{Precision} = \frac{\text{(number of relevant documents retrieved)}}{\text{(number of retrieved documents)}}$$

$$\text{Recall} = \frac{\text{(number of retrieved documents)}}{\text{(number of relevant documents)}}$$

Figure 2.3: Precision and Recall

Therefore, when designing information retrieval systems the Precision and Recall values are important to evaluate quality of search engines according to user preferences. Because, if Precision increases then this means user gets more relevant documents from retrieved documents. So, the irrelevant documents are further eliminated. On the other hand, if Recall increases then this means that users get more documents among the relevant documents. So, the quality of relevant documents in favour of the users is improved.

According to the above described facts, for instance if a search engine only results limited number of documents, it is necessary and feasible to improve Precision values of the search results. In this way, the user will retrieve the most relevant documents. The rest probably is irrelevant to the user and he does not need to read them. But, if the user needs or wants to learn all documents retrieved from an IR system, so the Recall parameter should be maximized. Then, the user gets all documents regardless of relevance or irrelevance degree in search results.

A classical Information Retrieval process is depicted in figure 2.4. At first, there is an information need by users and the need should be processed fast with correct results. Then, the user enters a query which contains some terms related to wanted documents' content. The query is executed by searching phase and ranked search results are returned from the IR system. Finally, the results are shown to the user by a user interface application. In searching process the query terms are matched with terms stored in an index, so indexing process an important role in IR. Because, well-implemented and designed index and index structure will give a better and faster search results.

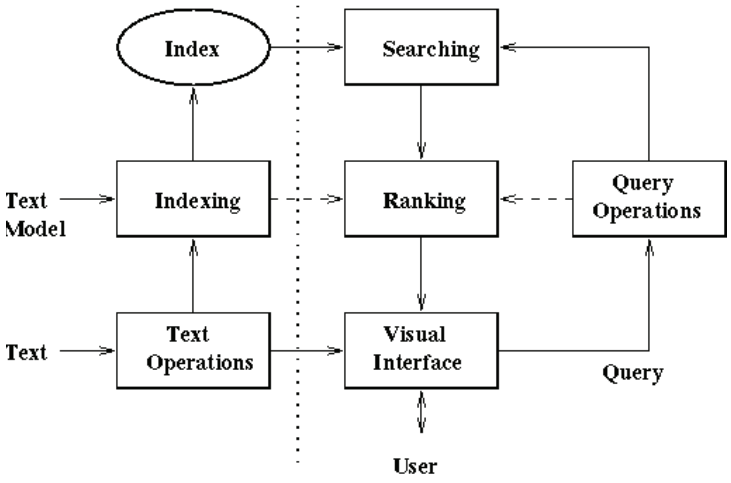


Figure 2.4: Information Retrieval Process [BR99]

2.3.1. Information Retrieval Models

In information retrieval environment, there are three main mathematical models [BR99] those describe fundamentals of indexing, searching, weighting, ranking, providing queries and documents evaluation. These models are important to understand what lies under indexing and searching functions.

The fundamental models are:

- Boolean model
- Vector space model
- Probabilistic model

Boolean Model:

Boolean model is known as the first information retrieval model. It uses simple algorithms like simple match in searching and relies on the use of Boolean operators (AND, OR, NOT). When searching documents, user can only enter queries like 'Java AND Swing', 'Java OR Swing' or 'Java AND (NOT Swing)'. This results exact matches for query terms in the index meaning 'yes' or 'no'.

Furthermore, in Boolean retrieval model all documents have same weighting. There is no term weighting, so it does not support ranking of indexed documents, either. This causes size of retrieved results to be either too large or too small. Also, the user must know the right term that he is searching. Therefore, nowadays it is inflexible and insufficient for modern and big information retrieval systems. In order to overcome these problems, fuzzy operators are implemented. Fuzzy operators provide more accurate and close search results than Boolean operators.

Vector Space Model:

Because of the above mentioned restrictions in Boolean model, an improvement was done and Vector Space model was introduced. The key idea in Vector Space model is representation of everything (documents, fields, terms or queries) as a vector in a multi-dimensional space. The representation of a query and two documents according to their term weights in a vectoral space is shown in figure 2.5.

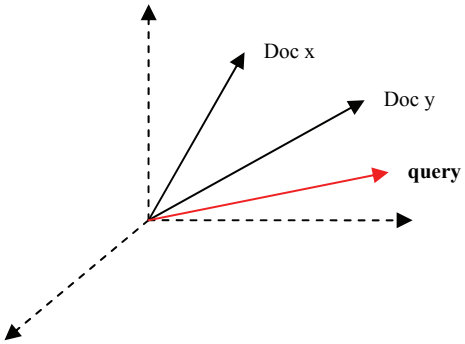


Figure 2.5: Vector Space Model

In this model term weighting, calculating similarity of documents, ranking of documents can be realized. In information retrieval process, the query vector is compared with other document vectors by calculating the cosine angle between the query and the document. In figure 2.5, the angle between document y and the query is smaller than the angle between document x and the query. This means that document y is more similar than document x by calculating the similarity using the similarity equation in figure 2.6 [LCS97], therefore the document y is more relevant to the entered query.

The similarity equation (figure 2.6) [LCS97] calculates the similarity value between a document $\boldsymbol{D_i}$ and a query $\boldsymbol{Q}$, where;

$\boldsymbol{w_{Q,j}}$ is the weight of term j in the query,
$\boldsymbol{w_{i,j}}$ is the weight of term j in the document i, and
the denominator is called the normalization factor.

$$sim(Q,D_i) = \frac{\sum_{j=1}^{V} w_{Q,j} \times w_{i,j}}{\sqrt{\sum_{j=1}^{V} w_{Q,j}^2 \times \sum_{j=1}^{V} w_{i,j}^2}}$$

Figure 2.6: The similarity equation

The similarity measurement is important in Vector Space model, because it used to retrieve documents according to a query and provide ranked results. So, the ranking is done after computing the similarity values between all vectors and the query. This results more accurate and relevant outputs with respect to Boolean model. Nowadays, most of the information retrieval services are based on Vector Space model (more information about information retrieval services and used models is given in section 3).

When we summarize the Vector Space model processes, there are three main steps in order to implement an information retrieval service. They are indexing, term weighting and ranking.

The first step is document indexing. It analyzes the documents (extractions of stop words or common words, stemming, synonym checking, filtering etc are applied to the documents). As a result, terms used in searching are produced. The indexing details are explained in section 4.1.2.

The second step is weighting of indexed terms. Here two parameters are important that are 'term frequency' (tf) and 'inverse document frequency' (idf). 'Term frequency' is the number of occurrences of term in a document; on the other hand 'inverse document frequency' is the measure of occurrences of term in all documents.

idf is calculated with "***log(N/f)***"

Here N is the total number of documents and f is the occurrence of term in whole documents. As a result, weighting of each terms is calculated as (tf * idf).

Finally after indexing and weighting, the similarity function (figure 2.6) is used for ranking the documents. According to the given query the similarities of each documents related to the query are calculated. As a result the ranking output, showing from most similar to least similar documents, is produced.

Latent Semantic Indexing:

Latent Semantic Indexing (LSI) is the variant of Vector Space Model. One of the features of Vector Space Model is the term or document vectors are independent from each other. The space matrix of documents, terms or queries are normally too large. Therefore, LSI aims to reduce the space matrix for indexing documents. So, the reduced space matrix probably gives better search results.

It uses Singular Value Decomposition (SVD), a dimensionality reduction technique, in order to determine uncorrelated, insignificant document or term vectors. So, LSI procedure is generally used for identifying synonym (words that have same meanings) and polysemy (a word that has multiple meanings) between documents. For example, if two documents have 'car' and 'vehicle' terms, then it can be concluded that these documents are relevant to each other. So, during searching with a query 'car', results also contain the document that contains 'vehicle' term.

LSI is a useful method, because it improves the accuracy and relevance of search results. But its main disadvantage is that it is computationally expensive especially in large matrices. Also, it consumes more time for comparing all vectors and reducing matrix dimensionality.

Probabilistic Model:

Probabilistic model is the newest model and rarely used in information retrieval services. It is based on the calculating the probability of relevance for retrieved documents. This is done with relevance feedback process. Relevance feedback holds the information about how relevant the retrieved documents are to a user during the search operations. It stores statistical information of relevant documents by starting from initial assumption. After the initial assumption the probability of relevance is improved during feedback of search processes.

Probabilistic model is more time and resource consuming process than other models. It claims that information retrieval process is uncertain, so there is no information that a correct match will occur in the retrieved results. In order to increase the accuracy of search results it calculates probability of relevance for the documents by using the equation **P(relevance/document)** with Baye's Rule. The calculation details of P(relevance/document) can be read in [Rij] - in chapter 6.

2.3.2. Search Engines (Full-Text)

Search engine is a program that searches for documents and lists retrieved results to clients. They became important and inevitable for Content Management Systems with the growth of data. Nowadays, most search engines are implemented to search through the whole internet and web pages or documents (e.g. Google is a famous web search engine).

There are three main parts in search engine applications which are document browsing (known as crawler or web spider), index and searcher interface. As the name implies, crawler's job is to find and gather as many document as possible for Information Retrieval. After that, those documents are ready for indexing. The indexing operation processes various analyses (see section 4.1.4) and outputs an index that is used for retrieving data. The index basically contains field and value pairs that identify contents of related documents. The index structure is further explained in section 4.1.2. Finally, after the index is created, there should be a search interface in search engines. This interface handles user inputs and shows retrieved results appropriately.

Early search engines did not use indexing; they simply looked for document titles with linear search. This process was slow and user needs to know at least some right words from title in order to retrieve some results. But, today most search engine application use index structure. This results in more accurate and relevant search results with various developed algorithms, models and methods.

On the one hand, full-text search engines mean that they analyze all textual content of any data and index them. So, they can only manage textual data. This is why they are called "full-text". For non-textual data, various document parsers are developed. They convert any type of documents into textual representation like XML to text, HTML to text or PDF to text. Then they able to index them and use in search engine applications.

In search engines there are some criteria which determine the quality and power of search system. The first one is the relevance of retrieved documents depending on user requests. Users generally want to retrieve data that match best during searching, because there are lots of documents and the time is restricted. It was observed that people mostly pay attention to first documents in search results (in some cases, the most relevant document for a user can be in lower rank, it might be overlooked), so the relevance is important in searching for documents in favour of the users.

The second one is the popularity of documents. This is widely known as ranking of documents. The popularity is assigned to documents by calculating weights of document terms. According to the document weights, the search results are ranked from highest to lowest values and listed.

The last criterion is the location of data. The idea in 'location' is to know where original documents reside (for example, the file URL in internet or the path in a file system). This checks the availability and reliability of retrieved results. Because, in user interface applications mostly a descriptive small size of text for the documents or a link where the original text exists is shown, so the location information is important too.

In this thesis, a full-text search engine application is implemented using Lucene as a search engine library. Then, it is integrated with CCMS, so it uses the data from CCMS database for full-text indexing and searching. The design and implementation of the full-text search engine can be read in section 4 and section 5 accordingly.

Chapter 3

3. Information Retrieval Services

In recent years Information Retrieval became an important area because the size and number of information increase day by day. Besides this growth, people need information or documents in a quick way with most relevant ones which satisfy user needs. For this reason, wide range of Information Retrieval services is developed to provide robust search engine features.

This first group are complete applications that implement all Information Retrieval functionalities from indexing documents to showing them to front-end users. These applications are ready to use and integrate to user's system without having much programmer interference. For example, web crawlers, web search engines or commercial content management systems are developed in this way. Therefore, users or programmers generally do not know which algorithms, models or methods are implemented during Information Retrieval process.

The second group are Information Retrieval libraries which provide the fundamental indexing (section 4.1.2) and searching (section 4.1.3) functionalities. They work as Application Programming Interface (API) in developing applications. The search engine application is fully controlled and developed by programmers. Indexing documents, parsing contents, getting user inputs and showing search results are all programmed by the programmers. Therefore in this thesis and implemented search engine application, one of the Information Retrieval libraries, which is "Lucene" [LUC], is selected and used.

In subsections of this chapter, the best Information Retrieval libraries and their comparison are explained. The reasons for choosing Lucene Information Retrieval library are detailly described.

3.1. Information Retrieval Libraries

Information Retrieval libraries are search engine Application Programming Interfaces (APIs) that provide basic indexing and searching functionalities for the content management systems that have large size of data to be managed.

The most available search engines were developed commercially. Therefore, there are not too many free search engine libraries. Recently some new and open source Information Retrieval libraries developed and they are further in development. In this part, the well-known and widely used Information Retrieval libraries are selected and explained.

The selected libraries are Lucene [LUC], Egothor [EGO] and Xapian [XAP]. In general, they are open-source, free and full-featured libraries. Lucene and Egothor are based on Java, on the other hand Xapian is written in C++. The detailed features and differences like supported languages, Information Retrieval models, used file formats, indexing and searching algorithms are described in following section 3.2.

3.2. Comparison of Selected Search Engine Libraries

Firstly, the used programming languages are Java in Lucene and Egothor, whereas C++ in Xapian. Lucene, Egothor and Xapian search engine libraries are well-known, free and open source tools. They can easily be integrated into search engine systems. The summary of comparison results are shown in figure 3.1.

When the underlying technologies are compared between these libraries, they differ on selected Information Retrieval models (see 2.3.1). Lucene is based on Vector Space model (also includes Boolean model) and this model has powerful algorithms and methods as explained. Also, today it is mostly used model between various libraries or search engine tools. Egothor is based on Extended Boolean model that supports basic Boolean model queries like (AND, OR, NOT) and also implements some extensions like using fuzzy logic, fuzzy operators (e.g. similarity search). It also uses some Vector Space model methods for ranking indexed documents. On the other hand, Xapian is totally different from the two tools and based on Probabilistic IR model. Because of the probability search Xapian is the most complex library that was developed recently.

All the developed and available search engine tools can only handle and execute textual data, this is way they are called “full-text”. Because of this restriction lots of third party documents parsers are implemented. These parsers provide the conversion of different type of files into plain texts. Therefore Lucene, Egothor and Xapian support widely used file formats HTML, PDF, MS Word, XML and so on. Also, special document parsers for other types can be written and be integrated into IR applications.

When the supported languages are investigated, Lucene supports English, German, Russian, Chinese and Korean languages for analyzing documents and indexing them. Egothor states it has a Universal stemmer that can analyze any European language. Also, it is important to mention that no testing has been performed on other languages except English. Besides, Xapian supports total 12 languages (English and most European languages, Danish, Dutch, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, and Swedish.).

	Lucene	**Egothor**	**Xapian**
Programming lang.	Java	Java	C++
License	Free, open-source	Free, open-source	Free, open-source
IR models	Vector Space Model	Extended Boolean, Vector Space Model	Probabilistic Model
Supported files	Text, HTML, PDF, XML, MS Word	Text, HTML, PDF, PS, MS Word, XLS	Text, HTML, PHP, PDF, PS
Query types	Boolean, Fuzzy, Wildcards, Range	Boolean, Fuzzy, Wildcards, Range	Boolean, Fuzzy, Wildcards
Supported languages for indexing	English, German, Russian, Chinese, Korean	Universal stemmer: any European language	Total 12 languages: English and most European lang.
Weighting techniques	Similarity, Latent Semantic Indexing	Similarity	Probability of relevance, relevance feedback
Index structure	Inverted index, incremental	Inverted index, incremental	Inverted index, incremental
Index storing	RAM, disk	RAM, disk	RAM, disk
Database indexing	YES	YES	YES
Max index size	2^32 docs	2^64 docs	2^32 docs

Figure 3.1: Comparison of Selected Information Retrieval Libraries

The ranking process of documents is done with different techniques among these tools. Lucene's ranking system is based on weighting of terms. Also there are special functions (like boosting term values, normalization of weight values) used during the ranking. So, Lucene uses Vector Space model's similarity calculation. The mathematical definition and calculation of similarity can be read in section 2.3.1. Furthermore, Lucene performs Latent Semantic Indexing (LSI) too in order to reduce search matrix. Also Egothor uses Vector Space model's weighting and ranking techniques. It performs similarity ranking between documents to improve relevance of documents.

Xapian executes completely different method for ranking documents. Because it is based on Probabilistic IR model, it calculates probability of relevance for documents in ranking. In calculation, it uses relevance feedback technique (see Probabilistic Model) for further improvement of relevance in search results. These operations are time and resource consuming, so indexing times are higher than Lucene and Egothor.

The common features of the selected libraries are; they support Boolean operators (AND, OR, NOT), wildcard searching (*: zero or many characters, ?:only one character), fuzzy operation (~: similarity searching), range and phrase queries. Furthermore, they all implement incremental indexing. Incremental indexing means that when new documents are indexed they are stored in a new file and then the new ones are merged with old index. In this way, a search engine application can simultaneously update and search indexes. This an important feature for IR services, because there is no need to stop searching or making search engine off-line while indexing new documents. So, there will be no time lost for search engine users.

As explained previously these libraries support well-known file formats. Besides, they can also index documents from databases directly. They implement their own index structures. Generally they index document terms with field and name pairs. The index can be created both in memory (RAM) and in disk storage. When we compare the limit of index sizes, Lucene and Xapian support 32-bit operating systems whereas Egothor 64-bit. Therefore, there can exist 2^32 documents in an index using Lucene or Xapian. On the other hand maximum 2^64 documents can be indexed using Egothor.

Furthermore, Lucene implements optimized memory management in applications. In Java virtual machine (JVM) less java objects are allocated, but Egothor does not support this memory optimization yet.

As a result, Lucene is selected and used for the full-text search engine application in CCMS. The reasons for selecting Lucene library are as follows. Firstly, CCMS is developed with Java programming language and Lucene is also based on Java. Also, Lucene is a free and open source Java library and is widely used by programmers. It has more literature than other libraries which are observed in this project. On the one hand, it has an efficient German analyser that can index and search German documents.

Furthermore, Lucene provides a compound index structure (see 6.2.1 for details). Basically, in the compound index structure the indexed document results are merged into one index file. This provides faster searching results because of the minimized index file accesses. Lucene library has index tuning parameters (see 6.2.3) that enable to adjust the searching process according to users' system resources. Also, Lucene supports Latent Semantic Indexing (LSI) (see Vector Space Model in section 2.3.1) which minimizes the index size further. Finally, it has a feature which is optimized memory management that allocates less Java objects in memory.

Chapter 4

4. Design of Application

In this section, the design of the information retrieval application for Conceptual Content Management is explained in detail. As explained in previous pages, the application is developed in order to work as a module in CCMS. The application retrieves data from CCMS-database, indexes the retrieved data and searches them with user queries which entered by user interface program.

Therefore, firstly in design phase some decisions were made. They are, the programming language used in applications, the information retrieval service for implementing indexing and searching methods, the document parsers in order to find information that is stored efficiently, accurately and the language analyser according to the language of data stored in database.

The first subsection explains the selected programming language and the selected information retrieval library (Lucene) for this thesis. In this subsection, the most important parts in search engine environment, indexing and searching are described. This part also includes the analysis process which is important for applications to convert textual data into index structure and use it in searching.

The document parsers aim at handling the documents which are XML files different from normal textual data, so the parsing methods are necessary in order to catch required texts from these files for indexing and searching. On the other hand, the asset data can be any type and they can be parsed further. So, the different type of data can also be indexed and searched. After that, modules which are implemented and used in application are explained using UML diagrams. Finally, the application logic and the available functionalities are described in detail. Also, at the end of this part, the overall system structure can be seen for further understanding.

4.1. Selected Search Engine Library: LUCENE

The comparison of different search engine libraries and information retrieval services was explained in section 3.2. As a result, the Lucene information retrieval library, which is based on Java, free, open source and widely used API, was selected in order to perform indexing and searching mechanism in the application.

4.1.1. The Functionality of Lucene

Lucene is an information retrieval library that is written in pure Java. It provides core Application Programming Interface (API) for adding full-text indexing and searching functionalities into developer's applications. Therefore, it is not a complete framework that performs all methods for implementing a search engine. It helps programmers with indexing and searching functions to convert any type of data to textual presentation index them to an index file structure and search with given user queries. So, the application logic how to manage indexing, searching, getting user queries or representing them to front-end client belongs to the programmers.

In traditional or first search engines indexing is done by keywords and its represented text pairs determined by programmers. In this type of document retrieval user can only use Boolean queries (AND, OR, NOT) for searching and it has some drawbacks like depending on indexers, using only Boolean queries and being time consuming. But with the new mathematical models, the functionalities in full-text search engines increased. Apart from Boolean queries term, range, prefix, phrase, wildcard and fuzzy queries are supported. Ranking the documents according to occurrence of search terms, having a complex index structure for retrieving the data efficiently are further developments in full-text search. The supported queries are described in section 5.4.

As a result Lucene supports full-text indexing and searching mechanism that is popular, widely used and supported by developers.

4.1.2. Indexing

One of the main concepts in Lucene search engine API is indexing. Indexing is conversion of any type of data into searchable index format. The conversion is performed by analyzers. Firstly, documents are retrieved and all unusable texts like stop words, word -

suffixes or prefixes are discarded. The details about analysis process can be found in section 4.1.4. As a result of indexing process an index is created.

A Lucene index consists of Lucene Document class instances which defines the index documents. Each document contains Fields those consist of name and value pairs. A sample index is depicted in figure 4.1.

Lucene Index Structure:

Lucene index structure [LUC2] is known as inverted index. Inverted index means that the content of documents is analyzed and the important terms are indexed as field name and value pairs. Each field contains many terms that point to corresponding documents in the content management system. The inverted index facilitates retrieving documents from a system and is used by search engine application. So, the documents are searched in the fields and in their values.

As it is shown in figure 4.1., Lucene index consists of many Segments. A Segment is created when a heap of new Documents are created and indexed. So, each Segment has many Documents stored in it. The Documents consists of indexed Fields. As explained the Fields have the smallest parts in index structure which have name and value pairs. These Fields are used for calculating weights and ranking search results.

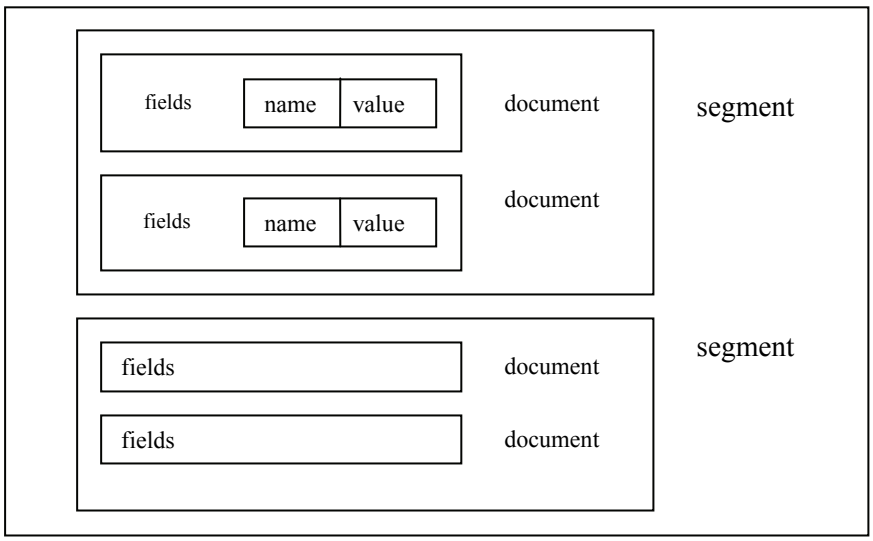


Figure 4.1: Lucene Index Structure

In indexing process the basic Lucene classes are [LUC3];

- IndexWriter
- Analyzer
- Directory
- Document
- Field

IndexWriter:

The IndexWriter is the main class in Lucene indexing operation. It creates the initial index file to desired path in your computer. As a parameter it takes the path, type of Analyzer and Boolean parameter (if Boolean value is true then it creates a new index from scratch, if false then it appends to existing index on that path). IndexWriter is the only class that has write-access to the index and using its methods users can add documents to the index for searching purposes

Analyzer:

The Analyzer class implements parsing of contents before creating the index file. It analyses the documents and discards text that is not useful in searching application. The analysis process is explained in section 4.1.4.

Directory:

The Directory class shows the place where the index is created. It has two subclasses which are FSDirectory and RAMDirectory. According to programmer preferences one can use both of them in indexing process. FSDirectory resides in a file system; on the other hand RAMDirectory resides in computer memory.

It is obvious that RAMDirectory has advantages over FSDirectory because when an index is on memory it is faster for indexing and searching documents than accessing to the index on a disk. But, when stop searching or exit from the application, the index on memory will be deleted. Therefore, it is important to store the index on a disk for future use. The evaluation of two Directory types can be found in section 6.2.2 in order to get a clear understanding.

Document:

The Documents class represents the fields in an index. It consists of fields that store name and value pairs. In Lucene, the original document or meta-data (such as title, date, author of that document) can be linked to these fields so the retrieval in searching is done efficiently. Also, in order to index any document it must be in a textual format or convertible to text.

Field:

The final core indexing class is Field; it represents the basic field structure in an index. Each of the fields shows information about their related documents and they are retrieved during the search operations from the index.

There are four types of Lucene fields which can be used according to application requirements. They are “Keyword”, “UnIndexed”, “UnStored” and “Text”. Here the field types, their features and used places are important in designing search engine because it directly affects searching process. Their overall features can be seen in the figure 4.2.

Field Type	Analyzed	Indexed	Stored
Keyword		*	*
UnIndexed			*
UnStored	*	*	
Text	*	*	*

Figure 4.2: Lucene Field Types and Features

The “Keyword” field type is used for indexing any text as it is written in the documents. This can be useful if one wants to index documents where original values are kept. For example titles, URL, dates, personal names or path of documents. As a result, with Keyword field type the data is not analyzed or tokenized but it is indexed and stored.

The “UnIndexed” field type is used for, as it can be understood from its name, indexing texts which will be neither analyzed nor indexed, but it is stored in the index. This is effective if one does not want to search a document directly but he wants to show the document in search results. The disadvantage of this field type is storing the documents as a whole, so if documents are large then the index size will increase.

The "UnStored" field type is the reverse of UnIndexed, it is analyzed and indexed but the original content of document is not stored in the index. This type is suitable for documents that have large size of contents, like HTML pages' bodies or any textual content of files.

The "Text" field type is mostly used when a user wants to analyze, index and store documents in his application. This type is useful for indexing small size of textual data like document title, description or subject.

The summary of indexing steps is shown in figure 4.3.

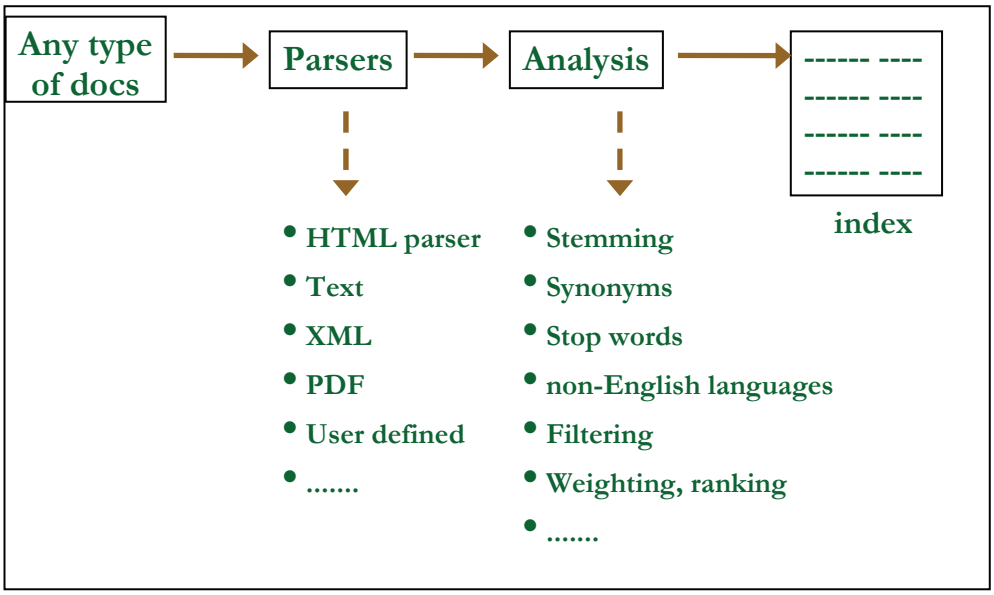


Figure 4.3: Main Indexing Steps

The indexing process starts with reading documents, in this case assets, and parsing their contents. The parsing is done for any type of documents that are convertible to texts. In general, Lucene can parse only textual data, but for other type of files (e.g. PDF, MS Word, XML, etc) there are specific document parsers. The list of Lucene document converters can be found in reference [LUC]. Also, the users can implement their own document parsers by converting any type of data into text.

In the implemented search engine application, the XML data are converted to texts. After parsing the assets, the analyzing phase starts. The implementation of Analyzer is shown in section 5.5. The main steps in analysis are stemming, removing of stopwords or common words, synonym checking, weighting of documents and ranking the results. Finally, the analyzer process creates a Lucene index which is the fundamental structure for searching mechanism.

4.1.3. Searching

Searching process is the second main step in information retrieval services. After indexing the required documents, in order to search them the searching methods should be developed. User queries are entered to an application, the queries are parsed by the searcher parser, and then hits are returned from the stored index and used for showing results to the user.

In Lucene the main search classes are [LUC3];

- IndexSearcher
- Query
- Term
- Hits

IndexSearcher:

As it is explained in previous section IndexWriter is used for indexing the documents, on the other hand IndexSearcher is used for searching a document from an index. It opens the index in read-only mode and uses its methods in order to return search results. Then, the results are ready for output, listing or sorting.

Query:

This class is used for defining user queries. There are lots of query types in Lucene which are BooleanQuery, FilteredQuery, MultiTermQuery, PhrasePrefixQuery, PhraseQuery, PrefixQuery, RangeQuery, SpanQuery and TermQuery. All these types of queries can be used in searching by creating manually or the Lucene QueryParser class can automatically fetch and understand in which type the user query belongs to. The Backus-Naur form (BNF) of Lucene query grammar is as follows [LUC3]:

```
Query  ::= ( Clause )*
Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")" )
```

Here the <TERM> describes in which index field will be terms are searched like “Title: Java”. (+) indicates the clause is contained and (-) indicates the clause is not contained in search criteria.

Term:

The basic unit in a search query is the Term class. It represents any text in a document while searching. The constructor has two parameters, one is the field in which the text will be searched and the other is the text itself. It is used for constructing a user query.

Hits:

After the construction of queries with Query and Term classes, the IndexSearcher class retrieves searched documents from the index. The matched documents are pointed by Hits class. The results come out as a ranked list. So, by implementing Hits class, the user gets searched documents, their scores and total number of documents.

4.1.4. Analysis

Analysis is the process of converting document texts into fundamental and indexable terms. Here the tokenization steps happen which are stemming, discarding stopwords, normalization, lemmatization and removing common, unuseful words from the document. Also, weighting and ranking of document terms is done in analyses part. Stemming produces the root of the words. The stopwords which are “and, or, but, not, then” and etc. are extracted. Normalization means to lowercase the text. Lemmatization is similar to stemming that produces basic tokens from the texts by normalizing words into the headwords. For example, the lemmatized form of the words “writing” and “written” is “write”.

Furthermore, in Lucene there exists different type of analyzers. They are GermanAnalyzer, RussianAnalyzer, SimpleAnalyzer, StandardAnalyzer, StopAnalyzer and WhitespaceAnalyzer. So, it is important to choose the right analyzer for the applications. In this thesis the content of documents are in German, therefore in application “GermanAnalyzer” is used. The usage of GermanAnalyzer in the application can be read in section 5.5. The following results show how different types of analyzers provide different outputs during analyzing of texts:

```
Analyzing "Analysis is the process of converting texts into terms."

WhitespaceAnalyzer:
   [Analysis] [is] [the] [process] [of] [converting] [texts] [into] [terms.]

SimpleAnalyzer:
   [analysis] [is] [the] [process] [of] [converting] [texts] [into] [terms]

StopAnalyzer:
   [analysis] [process] [converting] [texts] [terms]

StandardAnalyzer:
   [analysis] [process] [converting] [texts] [terms]

Analyzing "STS&TUHH - sts@tu-harburg.de"

WhitespaceAnalyzer:
   [STS&TUHH] [-] [sts@tu-harburg.de]

SimpleAnalyzer:
   [sts] [tuhh] [sts] [tu] [harburg] [de]

StopAnalyzer:
   [sts] [tuhh] [sts] [tu] [harburg] [de]

 StandardAnalyzer:
   [sts&tuhh] [sts@tu-harburg.de]
```

Figure 4.4: Comparison of Different Analyzers in Lucene Library

WhitespaceAnalyzer divides texts according to whitespaces in the text. So, each part is indexed as it is written as shown in the above tables.

StopAnalyzer firstly divides texts at nonletter characters, and then lowercases the letters. Finally it removes the stopwords which belong to the used language (e.g. English stopwords or German stopwords).

SimpleAnalyzer is similar to the StopAnalyzer, it divides texts at nonletter characters and then lowercases them but it does not remove stopwords from the texts.

StandardAnalyzer can be thought as a composition above explained analyzers. It performs all operations which Whitespace, Stop or SimpleAnalyzer do. On the other hand, it executes special operations according to the related language grammar. It can recognize abbreviations, e-mail addresses or special words (For example P&G has '&' character in its letters and StandardAnalyzer can efficiently index it as 'P&G' so it can be queried with the term 'P&G'. This is very important when indexing such words in order to search them correctly) and etc. As a result StandardAnalyzer is the mostly used analyzer in indexing and searching.

Lucene's weighting equation:

It is important to understand how Lucene's scoring algorithm works. It performs various operations in order to index and rank documents. These weighting values determine which documents are relevant to a given query. The score values are between 0 and 1. If the highest score is greater than 1, all scores are normalized from that value. Therefore returned Hits values are always between 0 and 1, meaning that 1 is the most relevant document and 0 is the least relevant document according to the entered query. The Lucene's scoring equation for a query (q) and a document (d) is shown below in figure 4.5 [LUC3]:

$$score(q,d) = \sum_{t \text{ in } q} tf(t \text{ in } d) * idf(t) * getBoost(t.field \text{ in } d) * lengthNorm(t.field \text{ in } d) * coord(q,d) * queryNorm(q)$$

Figure 4.5: Lucene's Weighting Equation

In this equation;

tf(t in d) is the term frequency of the term (t) in the document (d).

idf(t) is the inverse document frequency of the term (t).

getBoost returns the boost value of term field in the document, which is calculated and set during indexing.

lengthNorm function calculates the normalization for the term field in the document.

coord(q,d) computes a coordination value according to the query (q) terms the document (d) has. In Lucene API [LUC3] states that, *"The presence of a large portion of the query terms indicates a better match with the query, so implementations of this method usually return larger values when the ratio between these parameters is large and smaller values when the ratio between them is small."*.

queryNorm(q) computes the query normalization score for the query which is the sum of square values of terms' weights (the terms are retrieved from the given query).

4.2. Document Parsers

If a programmer wants to index a document it must be in a textual representation or be convertible to text. Therefore, there is a need of document handlers in order to index non-textual data like MS Word, PDF, XML files. In this thesis the retrieved data was as an XML format therefore different XML document parsers are developed. The implementation of document parsers with examples can be found in section 5.3.

Lucene does not have built-in common document parsers like other search engine libraries. They focus on indexing and searching functionalities with the intention of developing information retrieval applications. But, there are third party tools that can be easily integrated into such common document type handlers. For example, in Lucene API users can parse XML data using Jakarta Commons Digester [JCD], PDF using PDFBox [PDF], HTML using JTidy [JTi], MS Word documents using Jakarta POI [POI] or using built-in Java Development Kit (JDK) parser. As it is mentioned previously, in this thesis the stored data can be retrieved as XML files from the CCMS System, thus the Jakarta Commons Digester parser tool is used to handle these data.

Basically, Commons Digester allows programmers to map XML contents into Java objects with defined Digester rules. The rules show how to map XML tags, add calling methods, start or end tags and setter/getter methods for retrieving element/attribute values. An example of XML tags mapping is shown in figure 4.6. It is seen from this figure that in Commons Digester the parent/child relations can be easily coded in an application. The implementation of these document parsers are described in section 5.3.

XML file for an Asset	**Mapping value in Java class**
<data> <xml-fragment id="1118259927750"> <dok:typ>Zeitungsartikel</dok:typ> <dok:datum>2005</dok:datum> <dok:titel>Title</dok:titel> …… …….. </xml-fragment> </data>	data/xml-fragment , id (parsing **attributes**) data/xml-fragment/dok:typ (parsing **elements**) data/xml-fragment/dok:datum (parsing **elements**) data/xml-fragment/dok:titel (parsing **elements**) ....

Figure 4.6: XML Tag Mapping

As a result, after defining the XML tag mapping rules and creating call methods, using getter methods, the attribute value of ‘id’, element value of ‘dok:typ’,‘dok:datum’ or ‘dok:titel’ and so on are retrieved and indexed by Lucene’s indexing methods.

4.3. Definition of Modules

This project uses CCM System in order to retrieve documents from the database which is explained in section 2.3. Basically, the CCM System provides assets (concept-content model) for storing and managing data. It is implemented with the help of asset definition language and compilers. Also, the assets are stored in a database.

The first module is GKNS Module [CCM] that was developed as a part of CCMS. This module provides the main methods for creating, adding, deleting and modifying assets. Also, there are various functions that retrieve the assets from the database as XML stream data.

In this thesis, Lucene Module was designed and implemented as a sub-module in CCMS above the GKNS Module. A depiction of these modules and their relations are shown in figure 4.7. Lucene Module instantiates the GKNS Module and inherits all its methods. Furthermore, Lucene Module uses the inherited functions and implements new methods in order to retrieve all assets efficiently from the database, index the retrieved assets using Lucene search engine library.

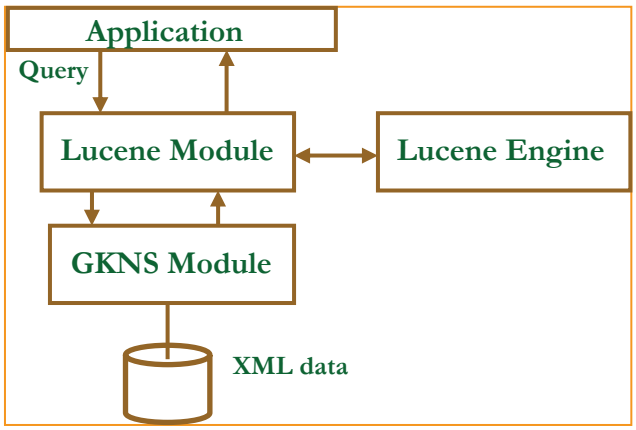


Figure 4.7: Modules in Conceptual Content Management System

At the top of GKNS and Lucene Module, the full-text search engine GUI is implemented. The search engine uses the Lucene engine and, realizes indexing and searching applications. The indexer application instantiates the Lucene Module and creates an index which stores all analyzed and indexed assets retrieved from the CCM System. On the other hand, the searcher application is programmed as a Java Client application that is a

user interface. This user interface gets user inputs, sends queries to the searching methods and retrieves wanted documents from an index. Finally, results are returned to the user and listed with details about the retrieved assets. The implementation details of the user interface are explained in section 5.7.

4.4. Overall System Structure

The parts used and implemented in this project are shown in figure 4.8. It consists of two main sections. They are the CCM System and the Lucene search engine application.

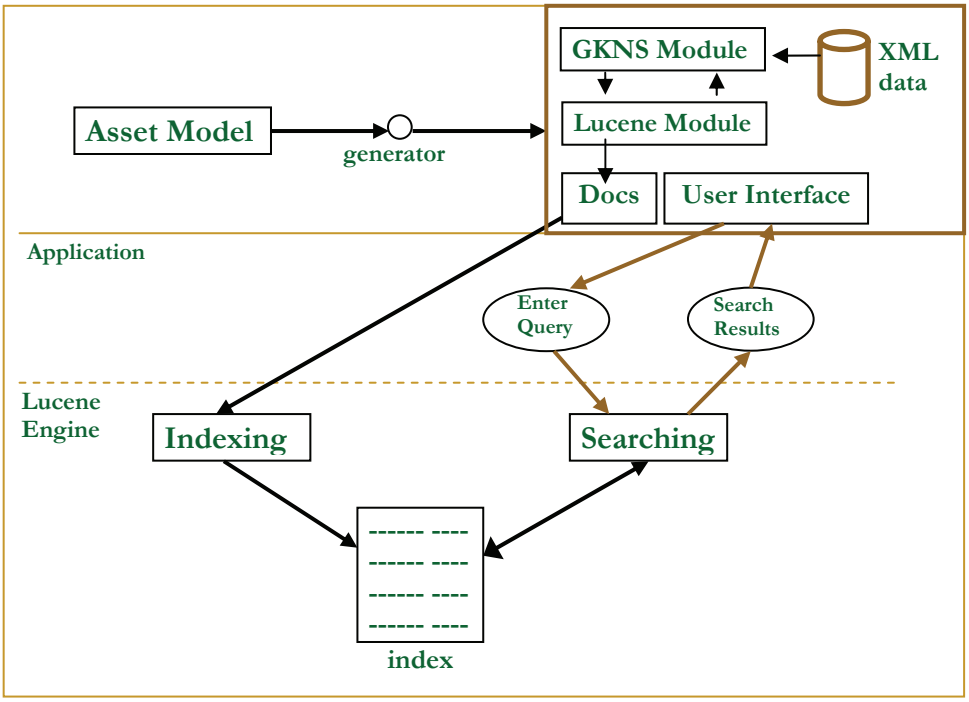


Figure 4.8: Overall System Structure

The first section contains previously developed applications which are Asset model definition, CCMS modules (GKNS module and Lucene module (with user interface) that is implemented during the thesis). The detailed definitions of modules are in section 4.3. Basically, GKNS module provides methods in order to create, delete, manage and retrieve data from a database. Lucene module implements interfaces for implementing a full-text search engine. It interacts with GKNS module and retrieves the data (the documents in the database are assets) that will be indexed and be searched. Therefore, searching functionality is performed by a user interface. The interface gets queries from users, triggers the searching operations and lists the returned search results. In the CCM system the data are assets. They are defined and the implementation is generated by the

Asset Definition Language (ADL). Assets consist of concept-content pairs, then this asset models are created by a generator in order to form a CCM system. Also, concept details and content of assets are shown to the users.

The second part is the Information Retrieval application itself. It instantiates the Lucene module and uses the Lucene search engine API. After getting all assets from the CCM system using Lucene module methods, these documents are sent to the indexer application. The indexing process analyzes the assets and creates an index in a file system. On the other hand, the searcher application handles user queries from the user interface and retrieves document matches from the index. Then, the search results which are ranked and showing the most relevant documents are listed.

Chapter 5

5. Implementation

In implementation part Lucene Module, indexer, searcher and document parser applications are developed. All these classes form the 'de.tuhh.gkns.informationretrieval' package in the Conceptual Content Management System (CCMS). Also, it interacts with other packages in CCMS which performs many methods for asset modeling. The overall structure of packages, the main classes contained in and their relations are shown in package diagram of CCMS in figure 5.1.

The implemented package classes perform the indexing and searching operations for CCMS. The subsection 5.1 describes the Lucene module and indexing of assets and their content in CCMS. The searching process is explained in subsection 5.2. The implementations of document handlers (like FundHandler, NachlassHandler) are explained in subsection 5.3. Furthermore, the query parsers supported by Lucene library are explained in 5.4 and the Analyzer used in the application is explained in 5.5. The subsection 5.6 shows the application logic and functionalities in indexing and searching process. Finally, the full-text search engine user interface and its features are shown in 5.7.

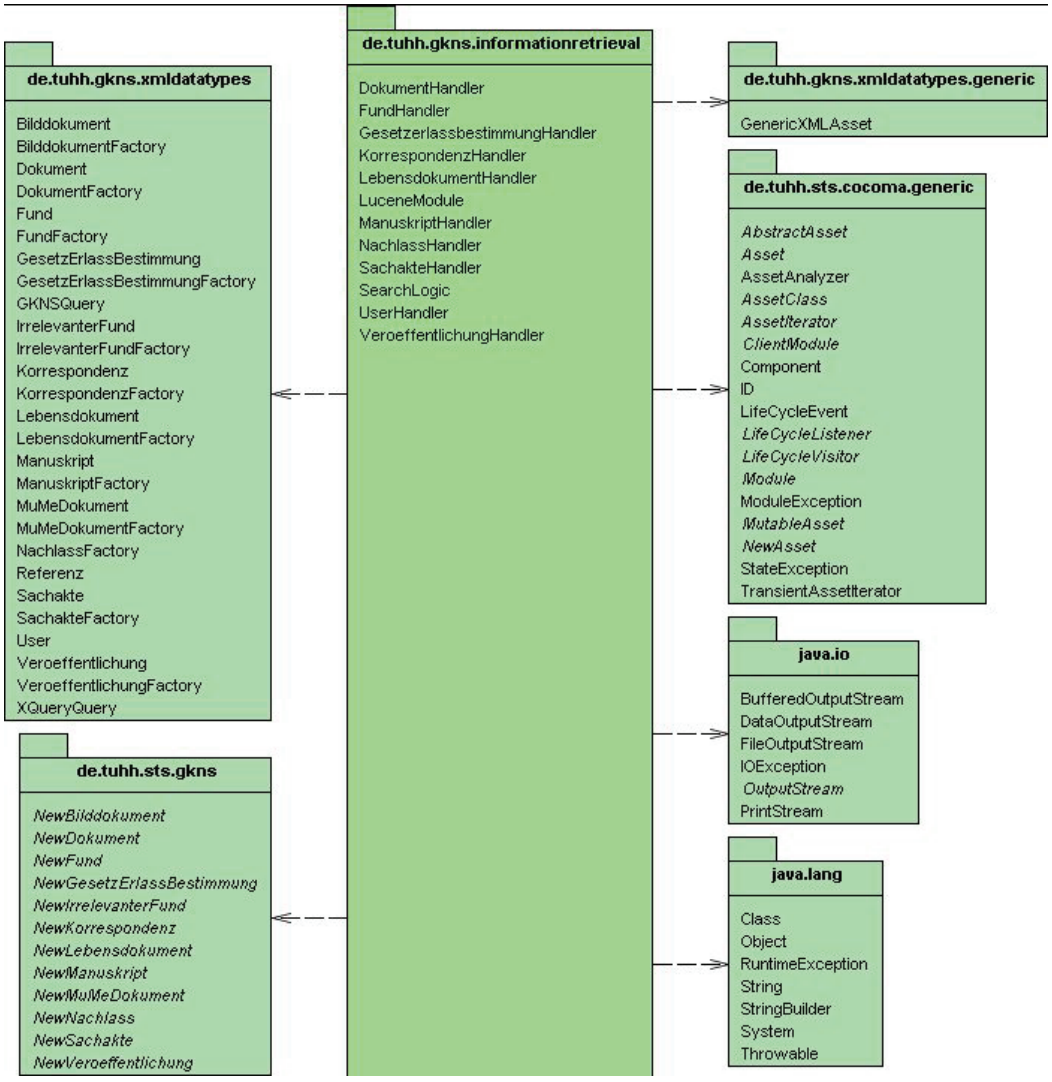


Figure 5.1: Package Diagram for Conceptual Content Management

5.1. Lucene Module - Indexing Process

The indexing of assets in CCMS is done by Lucene module's functions. Basically, Lucene module implements the ClientModule interface from CCMS and extends with its own methods in order to realize Information Retrieval in the system. The functions in Lucene module for indexing operations are:

```
start( )
stop( )
createInitialLuceneIndex( ), createInitialLuceneIndex (String)
retrieveAssets( ),
retrieveAssets(AssetClass, XQuery),
retrieveAssets(AssetClass[ ], XQuery)
indexAssets( ),
indexAssets(AssetClass),
indexAssets(AssetClass[ ])
getModule( )
getLuceneDirectory( )
```

Figure 5.2: The Main Indexing Methods in the Application

start( ): This method activates the created module.

stop( ): This method deactivates the started module in an application.

getModule( ): It used to retrieve the LuceneModule in order to reference it and use its methods in different applications.

getLuceneDirectory( ) : It returns the path of index from the file system which will be used in searching application.

createInitialLuceneIndex( ), createInitialLuceneIndex (String):

These methods are the starting point for creating an index for the Information Retrieval application. These methods create an initial index in a file system with a default location or the path of index is specified by a String parameter. After that this index is used for adding new assets from the database during indexing process.

retrieveAssets ( ), retrieveAssets (AssetClass, String XQuery), retrieveAssets (AssetClass[ ], String XQuery):

This is one of the most important methods during indexing, because it retrieves the documents (assets) which will be indexed. retrieveAssets( ) method by default gets the all assets from the database. Also there are two more variations of this method. One has the 'AssetClass and XQuery' parameters, the other one has 'array of AssetClass and XQuery'

parameters. The programmer can specify with ‘AssetClass’ or ‘AssetClass[ ]’ which asset class or array of asset classes will be retrieved and indexed. For this reason, the ‘XQuery ’ parameter forms an XML query [XQu] in order to retrieve assets from the eXist [eXist] database. An example programming code is shown in following figure. The assets are later retrieved by the searcher application and Lucene queries which is explained in section 5.2

```
String queryKorres = "declare namespace gkns =
'http://sts.tuhh.de/gkns/dokumenttypen.xsd' ; " +
"<gkns:allAssetList>{ /child::gkns:*[local-name(.)='korrespondenz' ]}" +
"</gkns:allAssetList>";

String queryFund = "declare namespace gkns='http://sts.tuhh.de/gkns/dokumenttypen.xsd';
<gkns:allAssetList>{ /child::gkns:*[local-name(.)='fund' ]}" +
"</gkns:allAssetList>";
```

Figure 5.3: XQuery Examples from the Application

indexAssets ( ), indexAssets (AssetClass), indexAssets (AssetClass[ ]):

After retrieving the assets, the ‘indexAssets’ methods triggers indexing classes for assets according to their types. The asset specification files describe schema of different asset types. The schema definition of assets can be found in appendix. The retrieved and indexed asset types in search engine application are as follows:

Korrespondenz
Bilddokument
Dokument
Fund
Gesetzerlassbestimmung
Lebensdokument
Manuskript
Nachlass
Sachakte
Veroeffentlichung

Figure 5.4: Asset Types in CCMS

An example of ’Fund’ asset schema definition is shown below:

```
<xs:complexType name="Fund">
<xs:sequence>
        <xs:element name="typ" type="xs:string"/>
        <xs:element name="datum" type="xs:dateTime"/>
        <xs:element name="titel" type="xs:string"/>
        <xs:element name="erfassungsdatum" type="xs:dateTime"/>
        <xs:element name="bemerkung" type="xs:string"/>
        <xs:element ref="gkns:kommentar" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="gkns:mask" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="erfasserRef" type="xs:ID" minOccurs="0"/>
        <xs:element ref="gkns:referenz" minOccurs="0"/>
        <xs:element name="verschlagwortungRef" type="xs:ID" minOccurs="0"
maxOccurs="unbounded"/>
        <xs:element name="contentIds" type="xs:string"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
```

This schema defines elements and attributes for the Fund asset. The elements are type of asset, creation date, title, entry date, remarks, comments, writer reference, content id, etc. The attribute is the unique ‘id’ value for the created asset. As a result, this schema and its values forms the **concept** part of a Fund asset, whereas with ‘contentIds’ element refers to the **content** of the Fund asset (Asset = concept + content model).

The methods ‘indexAssets (AssetClass), indexAssets (AssetClass[ ])’ can be used to index a specific asset type or types accordingly. As a default, ‘indexAssets ( )’ indexes all type of assets that exist in CCMS.

Finally, for each type of asset there exist document parsers. Basically, these document parsers handle different type of assets, convert the asset contents into textual representation and use Lucene library functions in order to index all documents. The parsing details are explained in section 5.3.

Adding assets to the index:

Using the Lucene module the initial index is created and the assets are retrieved from the database as XML data, then these data are sent to specific data parsers. These asset document handlers do the main indexing operations for each type of assets. Firstly, the index directory is read, and then the analyzer is chosen. In this project the content of data is in German, so we use Lucene’s ‘GermanAnalyzer’ for analyzing the data and indexing. In order to write the outputs from the analyzer and index them, the constructor of ‘IndexWriter’ is called. IndexWriter is like a pointer to an index and used to add new documents. The code is shown below:

```
String indexDir = setDir;
Analyzer analyzer = new GermanAnalyzer();
boolean createFlag = false; // means append to existing index without recreating
// IndexWriter to use for adding assets to the index
fsWriter = new IndexWriter(indexDir, analyzer, createFlag);
```

Now we have the index and IndexWriter instance for adding new documents. Therefore the program should create Document objects for different type of assets. The below example creates documents for "Bilddokument" assets. Then, different type of fields are added to a Document object with 'add (Field.<type>(name>,<value>))' method. Some asset data are added as 'Keyword' type fields and the other are added as 'Text' type fields. For example, 'ids' are indexed as Keyword fields or 'titels' are indexed as Text fields. The differences between field types are explained in Field title of section 4.1.2. After adding all asset values to the Document object as fields, finally the document is added to the index with IndexWriter's 'addDocument(<Document>)' function. This process is repeated until the application finish indexing all assets.

```
Document assetDocument  = new Document();

assetDocument.add(Field.Keyword("id", asset.getId( )));
assetDocument.add(Field.Text("typ", asset.getTyp( )));
assetDocument.add(Field.Keyword("datum", asset.getDatum( )));
assetDocument.add(Field.Text("titel", asset.getTitel( )));
assetDocument.add(Field.Keyword("erfassungsdatum", asset.getErfassungsdatum( )));
assetDocument.add(Field.Text("bemerkung", asset.getBemerkung( )));
assetDocument.add(Field.Keyword("erfasserRef", asset.getErfasserRef( )));
assetDocument.add(Field.Keyword("contentIds", asset.getContentIds( )));
assetDocument.add(Field.Text("entstehungsort", asset.getEntstehungsort( )));
assetDocument.add(Field.Text("inhalt", asset.getInhalt( )));
assetDocument.add(Field.Text("umfang", asset.getUmfang( )));
assetDocument.add(Field.Text("autor", asset.getAutor( )));
assetDocument.add(Field.Text("beteiligtePersonen", asset.getBeteiligtePersonen( )));
fsWriter.addDocument(assetDocument);
```

5.2. Searching Process

The indexing process analyzes all assets in CCMS and indexes them. At the end, an index is created and now it is ready for executing search operations. The main function for searching in Lucene library is *IndexSearcher* class. This class enables in programming to point to the index and read data from it.

Users enter a query to the search engine, then terms in the query are parsed and according to the query type assets are retrieved. This is performed by *IndexSearcher*'s *search( )* function. Finally, 'Hits' data structure is returned to the program. The 'Hits' includes the search results in a ranked order.

Also, with various functions, in program the details of documents can be read and outputted. The main programming parts for searcher application are shown below.

```
// default index directory, you can change it with method "setDir()"
private File indexDir = new File("indexableXMLFiles\\index");
private static Hits hits;
private static Document doc;
……..
// Refer to the created Lucene index in the directory
Directory fsDir = FSDirectory.getDirectory(indexDir, false);
IndexSearcher is = new IndexSearcher(fsDir);

// parse the query 'q
Query query = QueryParser.parse(q, "contents", new GermanAnalyzer());
……
hits = is.search(query);
……
is.close(); // close the Index after search operation is completed
fsDir.close(); // close the directory
```

5.3. Document Parsers

The returned asset data from CCM system is in XML streams. As explained in design of application part (see section 4.2) the parsing of these XML documents are done by Jakarta Commons Digester [JCD]. Commons Digester is based on SAX parser for document parsing. In figure 5.5 an example of XML Asset content is given, this Asset represents the structure of 'Dokument' type asset with elements and attributes. As mentioned previously, the asset types are "Korrespondenz, Bilddokument, Dokument, Fund, Gesetzerlassbestimmung, Lebensdokument, Manuskript, Nachlass, Sachakte and Veroeffentlichung". Therefore for each asset type a document parser is developed.

```
<data>
<xml-fragment id="1118259927750" xmlns:dok = "http://sts.tuhh.de/gkns/dokumenttypen.xsd"
xmlns:gkns="http://sts.tuhh.de/gkns/dokumenttypen.xsd">
  <dok:typ>Zeitungsartikel</dok:typ>
  <dok:datum>2005-06-08T00:00:00.000+02:00</dok:datum>
  <dok:titel>dokuement example</dok:titel>
  <dok:erfassungsdatum>2005-06-08T00:00:00.000+02:00</dok:erfassungsdatum>
  <dok:bemerkung>kein Angabe</dok:bemerkung>
  <dok:erfasserRef>1000000007</dok:erfasserRef>
  <dok:verschlagwortungRef>s1363</dok:verschlagwortungRef>
  <dok:contentIds>1118691449953</dok:contentIds>
  <dok:entstehungsort>bremen</dok:entstehungsort>
  <dok:sperrvermerkFachlich>2005-06-08T00:00:00.000+02:00  </dok:sperrvermerkFachlich>
  <dok:sperrvermerkJuristisch>2005-06-08T00:00:00.000+02:00 </dok:sperrvermerkJuristisch>
  <dok:inhalt>Alles ist möglich</dok:inhalt>
  <dok:umfang>keine</dok:umfang>
</xml-fragment> </data>
```

Figure 5.5: Sample XML Asset Content

The function of document parsers is to handle this XML file and retrieve the XML element and attribute values like id = "1118259927750", dok:typ = Zeitungsartikel, dok:datum = 2005-06-08T00:00:00.000+02:00, dok:erfasserRef = 1000000007, dok:contentIds = 1118691449953 and so on.

The XML file has a hierarchy as follows:

data/

data/xml-fragment

data/xml-fragment/dok:typ

data/xml-fragment/dok:datum

data/xml-fragment/dok:titel

data/xml-fragment/dok:erfassungsdatum

.....

On the other hand, this XML structure is mapped to a Java class. The java code for 'DokumentHandler' class is shown in figure 5.6. There is a simple direct mapping between the XML file Java class.

The attribute 'id' is mapped as follows:

```
digester.addSetProperties("data/xml-fragment","id", "id" );
```

The elements <dok:typ> and <dok:datum> as follows:

```
digester.addCallMethod("data/xml-fragment/dok:typ", "setTyp", 0);
digester.addCallMethod("data/xml-fragment/dok:datum", "setDatum", 0);
```

As a result, the parsing algorithm works as follows:

1) According to the asset schema using the Commons Digester tools, the programmer provides the rules for XML matching patterns to the parser. In the above example the top-level element is <data>. <data> has several <xml-fragment> elements that describe the assets and their values. So, the parsing algorithm visits all assets recursively until the </data> is matched.
2) For each <xml-fragment> the parser reads its child elements like <dok:typ>, <dok:datum> etc. and retrieves their element values.
3) These values are assigned to class variables by setter methods (setTyp( ), setDatum( ), etc.). Also by getter methods these values will be retrieved in indexing processes (getTyp( ), getDatum( )).
4) If there is a new asset element <xml-fragment>, the algorithm returns to the first step until all assets are parsed.

```
……..
// instantiate Digester and disable XML validation
Digester digester = new Digester();
digester.setValidating(false);

// instantiate DokumentHandler class
digester.addObjectCreate("data", DokumentHandler.class );

// instantiate asset class
digester.addObjectCreate("data/xml-fragment", Dokument.class );

// set id property of asset instance when 'id' attribute is found
digester.addSetProperties("data/xml-fragment","id", "id" );

// set different properties of asset instance using specified methods
digester.addCallMethod("data/xml-fragment/dok:typ", "setTyp", 0);
digester.addCallMethod("data/xml-fragment/dok:datum", "setDatum", 0);
digester.addCallMethod("data/xml-fragment/dok:titel", "setTitel", 0);
digester.addCallMethod("data/xml-fragment/dok:erfassungsdatum", "setErfassungsdatum", 0);
digester.addCallMethod("data/xml-fragment/dok:bemerkung", "setBemerkung", 0);
digester.addCallMethod("data/xml-fragment/dok:erfasserRef", "setErfasserRef", 0);
digester.addCallMethod("data/xml-fragment/dok:contentIds", "setContentIds", 0);
digester.addCallMethod("data/xml-fragment/dok:entstehungsort", "setEntstehungsort", 0);
digester.addCallMethod("data/xml-fragment/dok:inhalt", "setInhalt", 0);
digester.addCallMethod("data/xml-fragment/dok:umfang", "setUmfang", 0);

// call 'addDokumentAsset' method when the next 'xml-fragment' pattern is seen
digester.addSetNext("data/xml-fragment", "addDokumentAsset" );

// now that rules and actions are configured, start the parsing process
……
DokumentHandler dml = (DokumentHandler)digester.parse(is);
```

Figure 5.6: 'Dokument' Type Asset Parser Codes

"Digester digester = new Digester();" instantiates the Digester class and defines the parser methods.

"digester.addObjectCreate("data", DokumentHandler.class);" instantiates which type of document parser is used by Digester functionalities. In this case, the type is "DokumentHandler" document parser class.

"digester.addObjectCreate("data/xml-fragment", Dokument.class);" instantiates which type of asset is going to be parsed in this parser. In this case, the type is "Dokument".

"digester.addSetProperties("data/xml-fragment","id", "id");" method is used for adding attribute variables to the parser. In this case, the attribute name is "id".

“digester.addCallMethod("data/xml-fragment/dok:typ", "setTyp", 0);” method is used for adding element variables of XML file to the parser. In this case, the element variable name is “dok:typ”.

“digester.addSetNext ("data/xml-fragment", "addDokumentAsset");” parser method triggers the ‘addDokumentAsset’ function when next <xml-fragment> element is reached. ‘addDokumentAsset’ function creates the Lucene Document object of the parsed asset and with Lucene’s IndexWriter class this asset is indexed. So, the asset is ready for searching.

“DokumentHandler dml = (DokumentHandler)digester.parse(XMLstream);“: Finally, after defining the parsing rules, the ’parse’ function starts parsing for assets from CCM system and also indexing them.

5.4. Query Parsers

In search engines, query parsers are used to understand user entered query expressions. Also, it determines and executes Boolean operators, fuzzy logic, wildcard operations or phrase searching. In Lucene search engine library the query parsing is implemented by QueryParser class. In general, the parsing is done with the static parse( ) method in the QueryParser class. The parse( ) method works as follows:

*public static Query **parse**(String query, String field, Analyzer analyzer)*
throws ParseException

Parameters:
query - the user-entered query expression.
field - the default field name for the query (the field must exist in Lucene index).
analyzer – it analyzes the query with respect to given Analyzer type and transforms it into computer understandable string.
If there is an error like wrong syntax, then a parse exception is thrown.

The parse( ) method returns Query object. In Lucene, the Query object then instantiates its subclasses according to the parsed query expression. Query class has several subclasses; each of them implements specific query types. They are;

- BooleanQuery
- TermQuery
- WildcardQuery
- PrefixQuery
- PhraseQuery
- PhrasePrefixQuery
- FuzzyQuery
- RangeQuery

BooleanQuery:

This is the classical query type and used in all search engine application. It has the logical Boolean operators AND, OR and NOT. Furthermore, BooleanQuery is also used for defining complex clauses with other query types. In Lucene, it is declared as;

BooleanQuery bquery = new BooleanQuery();
In order to add clauses, the 'add' method is used:
bquery.add(< add a TermQuery >);
bquery.add(< add PrefixQuery >);

The details of add() method is as follows:

*public void **add**(Query query, boolean required, boolean prohibited)*:
The required and prohibited parameters specify the clauses;

- required: This parameter determines that if it is true the query must match, else it is optional (the clause exists or not)
- prohibited: This parameter determines that if it is true the query must not match in searching, else it also optional.
- none: If both parameters are false, neither required nor prohibited, this means that the clause is optional. There must be minimum one match from the clauses in order to match the Boolean query.
- But, both of the parameters cannot be true (required and prohibited). It is meaningless and invalid in searching.

In order to implement AND query, the 'required' parameter should be true and 'prohibited' parameter should be false. If the operation is OR, then the 'required' and 'prohibited' parameters should be false. For NOT operation, the 'required' parameter should be false and 'prohibited' parameter should be true. In Lucene, the user can form Boolean queries with -, +, AND, OR, NOT operators.

Finally, in Lucene the maximum number of clauses that can exist in a Boolean query is limited to 1024. This can be changed with method 'setMaxClauseCount'. If the limit is exceeded, it causes 'TooManyClauses' exception in program. This limitation is designed to avoid performance degradation in searching.

TermQuery:

TermQuery class is used to find a specific term from the Lucene index. The term represents the smallest structure in the index. It consists of a field name and a value pair.

Therefore, firstly a term instance is created by Term class as follows:

Term term = new Term("contents", "Zeitungsartikel");

The Term constructor has a field ("contents") and a value ("Zeitungsartikel") parameters, then a TermQuery is created:

Query query = new TermQuery(term);

As a result, this query returns all documents that have "Zeitungsartikel" value in their fields.

WildcardQuery:

WildcardQuery is a handy query type that matches words in a document although there are some missing letters in an input. There are two wildcard characters used in Lucene library which are * and ?. * means zero or more characters and ? means zero or only one character in query expression. WildcardQuery is a costly operation so it can take longer than other query types. In order to decrease the processing time, in Lucene the wildcards * and ? cannot be used as a first character in a query (*ava or ?ava not allowed). It is also interesting to note that if a query ends with wildcard characters, it is automatically transformed to PrefixQuery in the application.

For example, a query:

m*t can find documents that contains terms 'meat', 'meet', 'met', 'mat' etc.

me?t can find 'meat', 'meet', 'met' etc.

me?t* can find 'meat', 'meet', 'meeting', 'met', 'method', 'metal', 'meter' etc.

PrefixQuery:

PrefixQuery is a very useful query type in searching. It matches all documents with a specified prefix expression. For example, the query expression "prog*" will search for documents starting with the prefix "prog". So, it can find 'programming', 'programmer', and 'program' etc. terms from an index simultaneously. As an input syntax "prog*" is translated to PrefixQuery by the QueryParser when it is entered. In programming;

Term term = new Term("contents", "Zeit");

In index the assets that have prefix term "Zeit" in their "contents" field are searched.

PrefixQuery prefix = new PrefixQuery(term);

PhraseQuery:

PhraseQuery is used to find a specific order of terms in a document. For example, if someone wants to retrieve data that contains the phrase "Java programming". In this phrase, there would be no other term between 'Java' and 'programming'. Because, by default the slop factor of PhraseQuery is set to zero. The slop factor shows the number of words allowed to exist between query terms. It can be set to different value by the method 'setSlop(int)', so one can determine how many words could be between the terms in query. An example of different slop values and their results are shown below:

If slop factor is equal 0	"Java programming"
If slop factor is equal 1	"Java <any word> programming"
If slop factor is equal 2	"Java <any word> <any word> programming"

PhrasePrefixQuery:

PhrasePrefixQuery is an extension of PhraseQuery. It is newly developed and so far not supported directly by QueryParser class. It will be used in such an expression as "find documents that have term 'java' and 'prog' as a prefix term". If we formulate this; "Java prog*", it includes both phrase and prefix queries.

FuzzyQuery:

FuzzyQuery is based on the fuzzy logic; it derives from the extended Boolean model. The main object is to find similar documents with respect to given query terms. The similarity of terms is determined by the Levenshtein distance algorithm [GS], it is also called edit distance. Basically, this algorithm finds the number of steps in order to transform term x to term y.

For example, x= neet and y = meat

1) start : neet
2) meet (n -> m)
3) meat (e -> a)
4) end : meat

So, the Levenshtein distance is 2.

FuzzyQuery is used in Lucene with the character '~' in queries like 'meat~' also search the similar terms 'meet', 'met', 'meets', 'seat', 'mate' and so on. Therefore, fuzzy query is very powerful searching process.

In Lucene, there is a variable called 'minimumSimilarity' that defines the min edit distance value in fuzzy query. The default 'minimumSimilarity' value in the Lucene library is 0,5 (it must be between 0 and 1). If the edit distance is less than this equation;

*length(term) * minimumSimilarity* ,
then it means that the terms are similar according to this parameter.

It works as follows; if two terms are 'logic' and 'magic', and minimumSimilarity is equals to 0,5. The edit distance between 'logic' and 'magic' is;
logic -> mogic -> magic, so edit distance is 2.

The value *length(term) * minimumSimilarity* = 5 * 0,5 = 2,5
As a result, edit distance = 2 < 2,5 means that the terms 'logic' and 'magic' are considered similar and it is possible to retrieve in search results with the fuzzy query 'logic~'.

Proximity Search:

Lucene also supports proximity searching. It is mostly used if the users do not know the exact words in a phrase or want to retrieve terms within a certain distance. For example, if someone wants to search for documents that contain terms 'java' and 'programming' but within 5 words in the documents, then the query is:
"java programming"~5

RangeQuery:

RangeQuery is a powerful query type, it can retrieve documents with range values (*start* TO *end*). The terms of documents in the index are listed lexicographically, so this feature provides efficient searching with range queries. The RangeQuery constructor:

public **RangeQuery**(Term lowerTerm, Term upperTerm, boolean inclusive)

The lower and upper terms specifies the range of searched terms. The third parameter inclusive defines either the lower and upper terms are included in searching or not. As a result, range query can be efficiently used in dates (daily, monthly or yearly ranges), keywords or identifier values.

An example of RangeQuery usage:

Term start = new Term("datum", "20050801");
Term end = new Term("datum", "20050831");
RangeQuery range = new RangeQuery(start, end, true);

Indexing Dates and Using in a Range Query:

It is problematic in Information Retrieval services to index dates, because the representations or structures of dates in programming differ (especially in databases) and it may not be handled properly. For this reason, Lucene provides a special indexing method for dates which is:

Field.Keyword(String, Date) or *Field.Keyword(String, String)*

In our indexer application, it used as follows:
assetDocument.add(Field.Keyword("datum", asset.getDatum()));

Using this method, the dates can be indexed with different formats like with only year (YYYY), month and year (YYYYMM) or day,month and year (YYYYMMDD). In our Lucene search engine application, the retrieved dates from the database are in String format, for example *2005-06-08T00:00:00.000+02:00,* generated from Java Date class. In indexing this date structure is parsed and day, month and year parts are extracted. Then, they are indexed in YYYYMMDD format. This is a useful structure, because in range queries users have lots of alternatives in searching documents with date values. The users can enter queries as follows:

year	-	datum:[2000 TO 2005]
year, month	-	datum:[200001 TO 200501]
year, month ,day	-	datum:[20000101 TO 20050130]

Query expressions in Lucene:

The implemented search engine application can handle the operations shown in figure 5.7. These queries are parsed by *QueryParser* class in Lucene library and translated to the suitable query types.

Query expression	Retrieves documents that contain….
Art	the term 'art' in the default field.
art history art OR history	the term **art** or **history**, or both of them in the default field (the default operator is **OR**)
art AND history +art +history	the terms **art** and **history** in the default field
typ:Bestellung	the term 'Bestellung' in the field name 'typ'
art –history art AND NOT history	the term **art** in default field and do not contain **history** term
title:art –typ:Bestellung title:art AND NOT typ:Bestellung	the term **art** in 'title' fields and do not contain **Bestellung** in 'typ' field
(art AND history) OR Bestellung	the terms **art** and **history**, the term **Bestellung** is optional, all in default field
"Albrecht Altdorfer "	the phrase **"Albrecht Altdorfer"** in default field
Absender: "Albrecht Altdorfer "	the phrase **"Albrecht Altdorfer "** in **absender** field
prog*	the terms like *program, programmer, programming* etc in default field (see WildcardQuery)
contents:prog*	the terms like *program, programmer, programming* etc in **contents** field (see WildcardQuery)
me?t	The terms like *meet, meat, met* etc in default field
Meet~	the similar terms to **meet** like meat, met, seat etc. (see FuzzyQuery)
Datum: [20050801 TO 20051215]	the dates between *01/08/2005* to *15/12/2005* in **datum** field (see RangeQuery, date format: YYYYMMDD)
"art history"~ 4	the terms **art** and **history** within four words of one another in a document (see proximity search)

Figure 5.7: Query Expression in Lucene

5.5. Analyser

The analyzing process is the most important phase in information retrieval applications. Basically, analysis, in Lucene, means converting textual data into smallest tokens named terms. These terms represents their corresponding documents and are used for searching documents from the index. So, an analyzer performs complex and various operations in order to produce documents terms. These operations are stemming of words, synonym checking, removing stop words (e.g. and, not, the, of, etc), discarding punctuation marks, lowercasing the texts also called normalizing and deleting common words.

Lucene provides different analyzers for languages, in our search engine application the texts are in German. So, the GermanAnalyzer class of Lucene is used for analyzing the assets and indexing them. GermanAnalyzer provides a default list that contains German stopwords. Also, users can add their own stopwords to this list that will not analyzed and indexed never. On the other hand, a developer can need a list of words that must not be analyzed but be indexed. This is known as exclusion list in Lucene, the user provides this list as a file to the analyzer. The second way for enabling exclusions is to use 'Field.Keyword (….)' function during indexing (details in section 4.1.2). It does not tokenize words but index them as it is written in documents.

The indexing results in this project, and studies done in analyzing texts and stemming them show that these algorithms are not complete and perfect. Especially, in German where the form of words is complex and has specific features (e.g. 'Umlaut'). The stemming algorithms for German and their results can be read in [Cau99].

When the Lucene library and the implemented information retrieval application are examined, it is also error-prone and has some weakness. The first disadvantage is the GermanAnalyzer lowercases all 'Umlaut' characters (ä to a, ü to u, ö to o) and changes the character 'ß' to 'ss' while indexing. For example, if there is a word 'Häuser' in a document. Then, the GermanAnalyzer will produce tokens from 'hauser'. This results problems while searching, because normally the query 'Häuser' will not produce a matched document in spite of existence in index. In order to solve this problem, at the beginning we parse the queries and handle 'ä' as 'a', so 'Häuser' will match the required document in the index. This process is also same for other 'Umlaut' characters 'ö' and 'ü'.

The second disadvantage in German grammar is that plural forms modify vocals in the middle of words and irregular verbs change the words completely. For example, 'Mund - Münder' (mouth) or 'essen - aß - gegessen' (eat). On the other hand, the words 'Eis' (ice) and 'Eisen' (iron) are different in meaning but they will produce the same term 'eis' or 'ei' while stemming these words (the suffixes 's' or 'en' are discarded from words). This occurs rarely in indexing process therefore it can be ignored. This mainly results from the suffixes and prefixes that exist in nouns, verbs and adjectives. The following example shows a stemming operation performed for a German sentence and the outputted tokens are listed.

Input sentence :
"Während die Standardsprache in den meisten europäischen Ländern aus dem Dialekt der jeweiligen Hauptstadt hervorgegangen ist"

The returned tokens after the GermanAnalyzer function is executed are shown below:
[wahr] [standardsprach] [meist] [europaisch] [land] [dialek] [jeweilig] [hauptstad] [hervorgegang]

For example if we look at the term **[wahr]**, it is important to notice that if a user enter 'während' as a query, there will be not match for this sentence. So, the user query must contain 'wahr' or similar to this word like 'wah*' or 'wah?'. Lucene search engine library works different than the normal internet search engines like Google. They search for documents that contain the query terms word for word, but in Lucene the smallest parts of words which are tokens are effective while searching information from the index.

As a result, the Lucene library and the implemented information retrieval application have some drawbacks. But, it works well if the query types are used correctly and this library is a free-open source tool that can be used in many search engine applications effectively.

5.6. Application Logic and Functionalities

The main functionalities that the full-text search engine provides are indexing assets and searching them. The original assets (concept and content pairs) are stored in a repository. The application logic, indexing the documents and searching process according to a given query, is explained in following paragraphs using sequence diagrams.

The first sequence diagram is for the indexer application and shown in figure 5.8. Firstly, in indexing operation the Indexer class is instantiated. The indexer instance triggers Lucene module (LuceneModule class) implementation. Lucene module has the main logical methods for retrieving assets from the database and indexing them.

Lucene Module gets all assets and for each type of asset it starts DocumentParser objects iteratively. For example if the asset type is 'Bilddokument', then the DocumentParser which is developed for 'Bilddokument' assets is called or if the asset type is 'Fund', then the DocumentParser for 'Fund' assets is called. This is also performed iteratively for the remaining asset types.

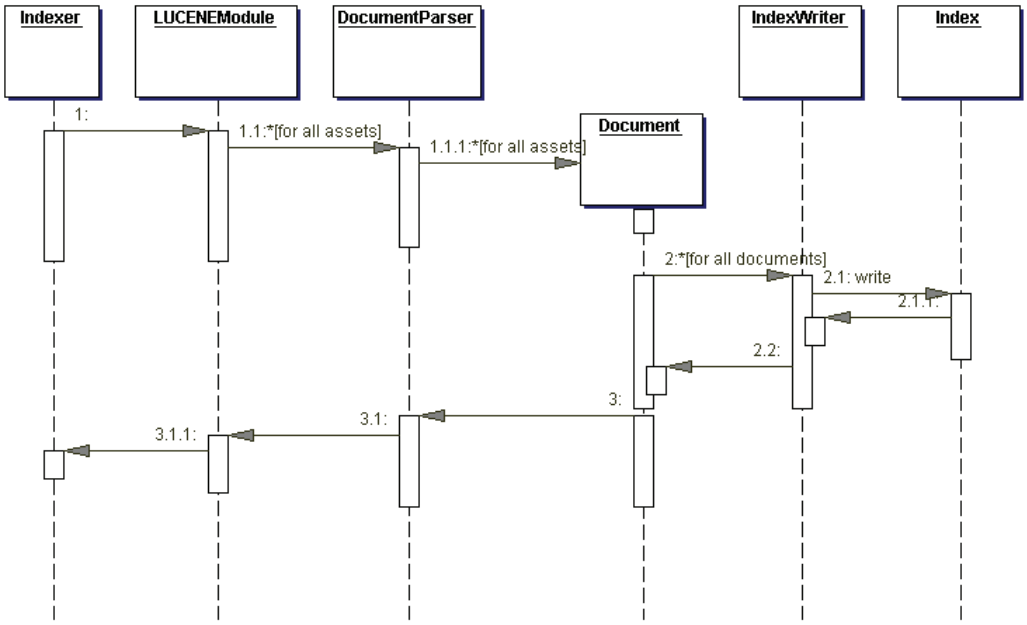


Figure 5.8: Sequence Diagram for Indexer Application

After all assets are parsed and analysed, the important terms are extracted from the asset data. For each asset a Document object is created. This Document object contains the terms which are generated by the document parsers. Then all Document objects are sent to IndexWriter object. IndexWriter performs the indexing operation.

The indexing is performed by writing all Documents into the Index. The Index is created by the program in a file system. This writing process into the Index is repeated until the entire Documents are finished indexing. Finally, the Index is optimized, closed and returned to Indexer application.

On the other hand, the searcher application provides searching assets from the CCMS system and outputting them as a list. The sequence diagram for searcher application is shown in figure 5.9. The searching process starts with initializing index directory and analyzer type (in this case, the assets are in German, so the analyzer is ‘German analyzer’). Also, the search query is retrieved from users.

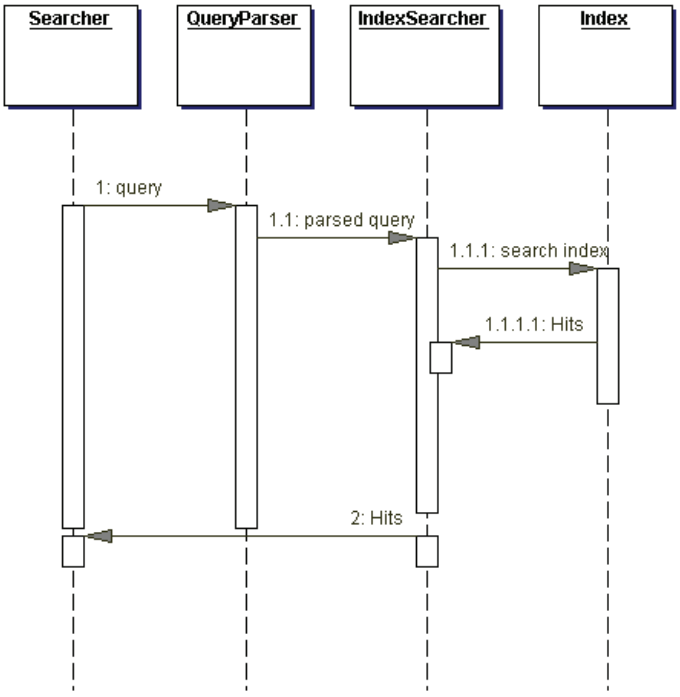


Figure 5.9: Sequence Diagram for Searcher Application

After the initialization, the user query is sent to QueryParser function for analyses. QueryParser determines type of query (e.g. Boolean query, Term query, etc.). The parsed query information is used by IndexSearcher class which implements search operations in given Index. The IndexSearcher triggers full-text search and the Index returns a 'Hits' object. It contains the search results retrieved from the Index. The search results are a ranked list of assets. Using 'Hits' class methods the asset information can be read and outputted to the user. Finally, the search results are sent back to the Searcher application and they are used for displaying content-concept parts of fetched assets.

5.7. User Interface

The application based on Lucene full-text search engine has a user-friendly and multi-functional user interface as shown in figure 5.10. It contains classical functions like text area for entering user queries, search button for triggering the information retrieval process and output list that shows search results retrieved from the index. Most of the search engines include these facilities.

The searching is executed by default field in the index. In this case, the default field name is 'contents'. It includes the largest terms indexed in it, so general searching can

be done with this field. Also using the combo box available in the user interface, the users can further narrow their search results resulting in more relevance documents. The combo box includes other field names that are more specific than 'contents' like 'titel', 'datum' or 'remark' etc.

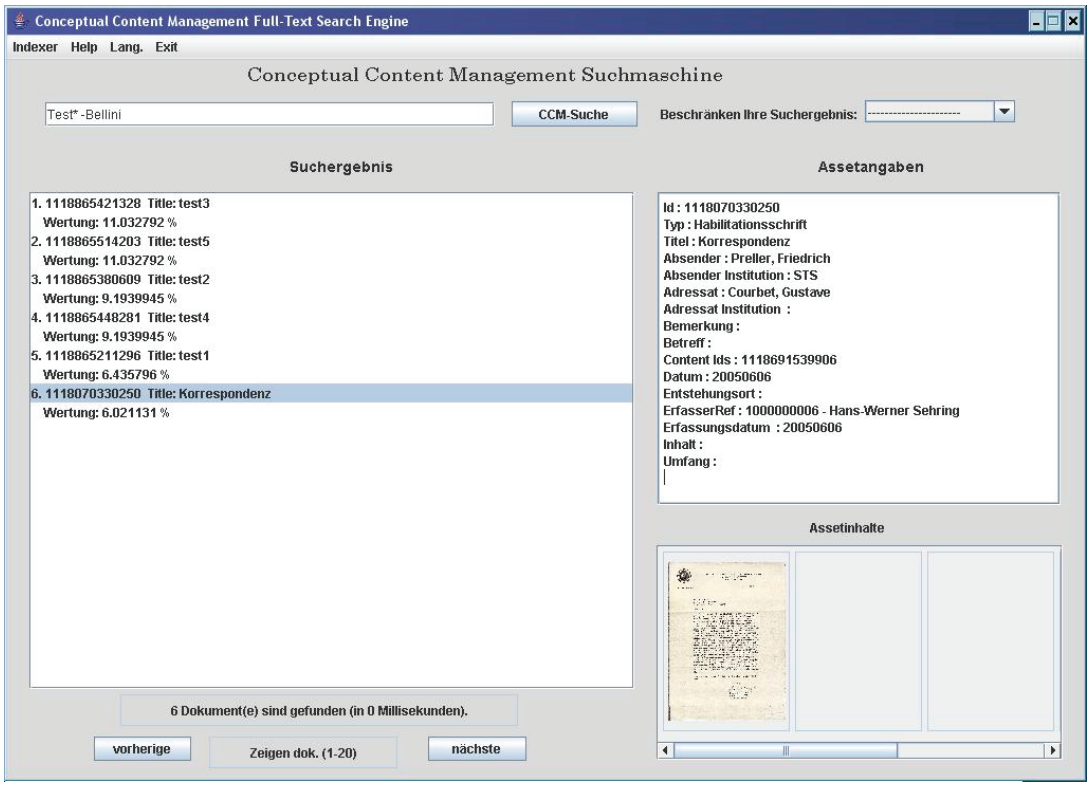


Figure 5.10: The User Interface of Search Engine Application

In developed full-text search engine the retrieved data are assets. Assets consist of concept and content parts. So, we introduced two new areas that show the conceptual data of an asset and its content as thumbnails. The thumbnails show images that belong to the content and they can be selected and be maximized to their original sizes (an example showing the content of an asset is in figure 5.11). The conceptual and content outputs are automatically changed according to selection made from the list of search results.

The output list shows the retrieved assets in a ranked order. On one page just twenty assets are listed, but using 'previous' and 'next' buttons users can navigate through all search results. Also, as further information, total time for searching documents and how many assets are retrieved are shown. In the output list, there is brief information about the retrieved assets which are identifier number, title and score (shows the relevance of an asset to a search query) of the assets. By clicking on a row in the list, it results in outputting the concept and content details of the selected asset.

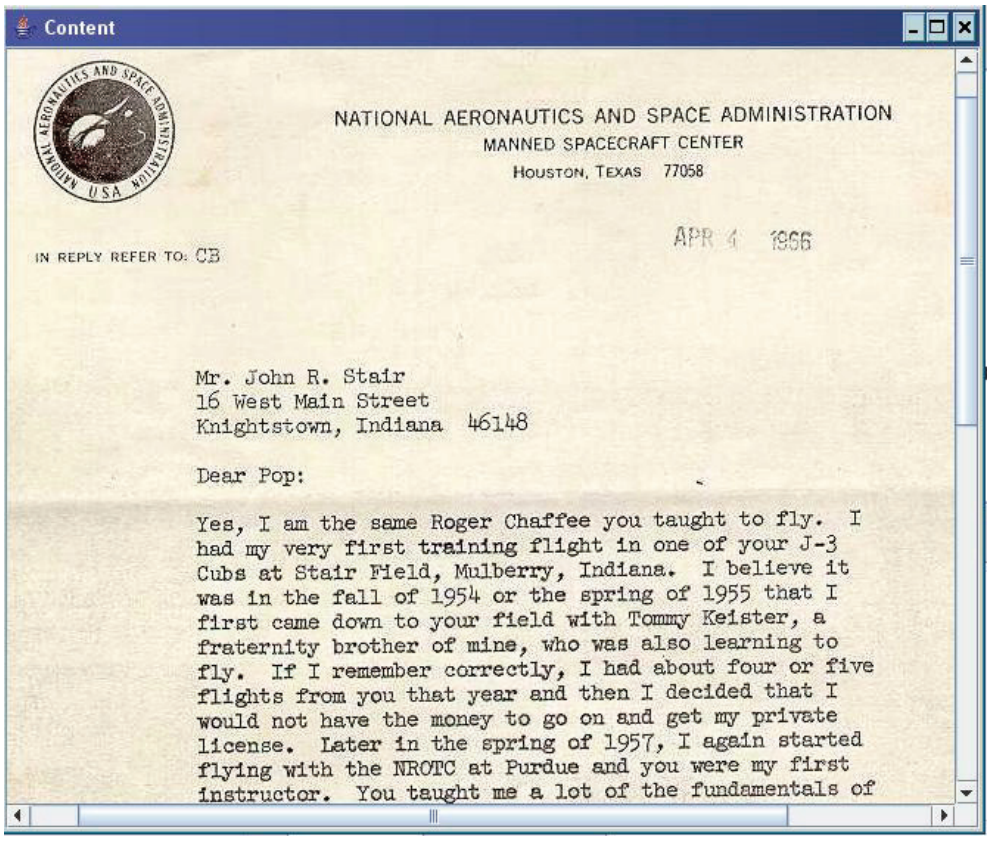


Figure 5.11: The Content View of an Asset

The conceptual part of the user interface (which is the 'Assetangaben' area) shows all attributes of an asset (the attributes are defined characteristics and relationships). On the other hand, the content area ('AssetInhalte') shows content of the asset which is closely connected with its conceptual part.

The Indexer User Interface:

The indexer application is developed in order to perform indexing operations automatically and easily. It has a user interface as shown in figure 5.12. The 'Retrieve & Index Assets' button triggers the methods for retrieving all XML data from CCMS's database and indexing them using Lucene search engine library. As a result, the index is created for searcher application.

Furthermore, the indexer application outputs some information about the indexing process. The information includes total time for indexing all assets, how many assets are retrieved from the database and indexed, and what are the available fields in the index. Also

if new assets are created or added to the database, the update can be easily performed. As a result, this application facilitates the indexing process with visual components.

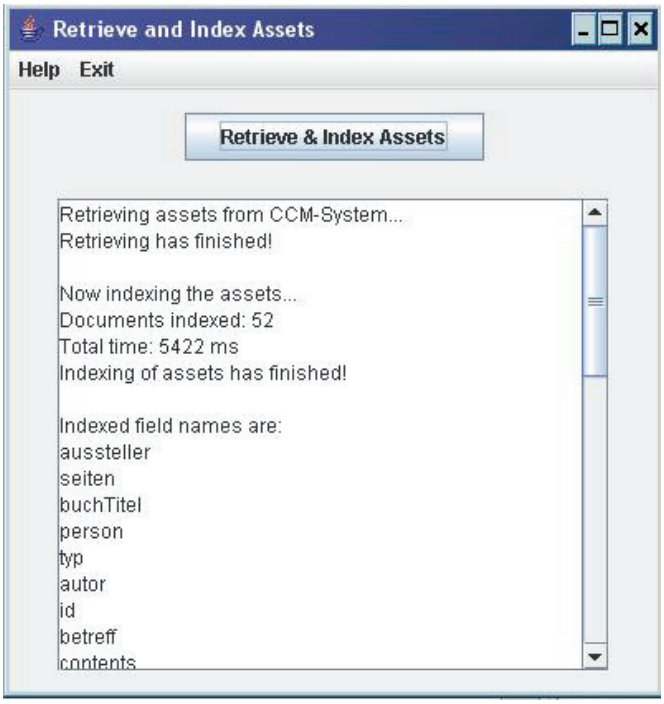


Figure 5.12: The User Interface of Indexer Application

Chapter 6

6. Evaluation of Results

6.1. Facilities for Conceptual Content Management

The developed search engine application provides indexing and searching methods for Conceptual Content Management System (CCMS). The CCMS does not have a built-in search engine feature. Therefore, the implemented program facilitates the retrieval of assets from the system.

The assets in a CCMS database could be large in number and size of files. So, searching for assets relevant to the users is required and important. Lucene library includes the basic information retrieval functions. The full-text search engine has an efficient user interface with visual tools for searching, listing result, navigating between search results, and outputting details of assets with text areas. So, retrieving required assets is easier and faster than looking them one by one. The search times take generally milliseconds. It is known that fuzzy queries take much more time than other query types, but it is observed that in the search engine fuzzy queries work as quick as other types. Also, the ranking and scoring features of information retrieval library provide the most relevant documents in a ranked order.

In CCMS the assets consist of concept and content parts as explained. So, there is a need for the search engine to show both parts in an efficient way in favour of users. The realization of concept – content monitoring is done by dynamic update of asset details and their content. If an asset from the result list is selected, details of the concept attributes and the content file associated with the concept are outputted automatically. All these features provide easiness with the purpose of executing information retrieval in a complex system like CCMS for users.

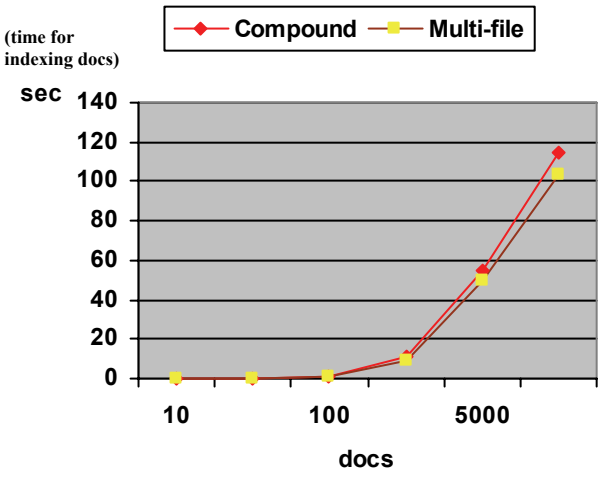
6.2. Test Cases

6.2.1. Compound versus Multifile Index

In Lucene there are two types of index structure [LIA04] compound index and multi-file index. In multi-file index, when new documents are inserted to an index, they are stored in a separate segment; this causes increase of files in an index structure. Therefore, multi-file index has more files than compound index.

Compound index type consists of three files; two of them are “deletable” file that shows the unused files in index and “segments” file that shows the segment names and their size. The third one contains the all indexed documents and their field values. In compound index all indexed files are merged into one single file. So, the number of files in the index is minimized.

The comparison results between multi-file index and compound index are shown in figure 6.1. The advantage of multi-file is the time for indexing documents takes less than compound file. Because, in compound file the indexed files are in addition merged into one single file. This can be suitable when the number of documents is large while indexing.



#docs	Comp. (sec.)	Multi-File (sec.)
10	0,172	0,078
20	0,313	0,250
100	1,046	0,970
1000	10,750	9,220
10000	115	103,5

Figure 6.1: Compound vs. Multi-File Index

On the other hand, the advantage of compound file appears in searching. Because, the total number of file accesses for reading data are minimum in compound index. In contrast, using multi-file index the file fetches increase because the program needs to open more files in order to retrieve required documents from the index. This is important while search time in an application is in consideration.

If the number of files opened during index operations are compared, the multi-file and compound index structure differ as follows: for example a search engine application uses 10 indexes, each index has 10 segments, each segment has 20 documents and each document contains 5 indexed fields. Then, in multi-file index case:

(10 indexes) * (10 segments per index * (20 docs per segment + 5 fields)) = ***2500***
files are opened during execution.

On the other hand, in compound index case:

10 indexes * 10 segment per index * 1 docs per segment = **100**
files are opened during execution.

The compound index opens considerably less files than multi-file index; therefore it consumes less system resources while searching. Also in some operating systems the number of files opened at the same time is restricted.

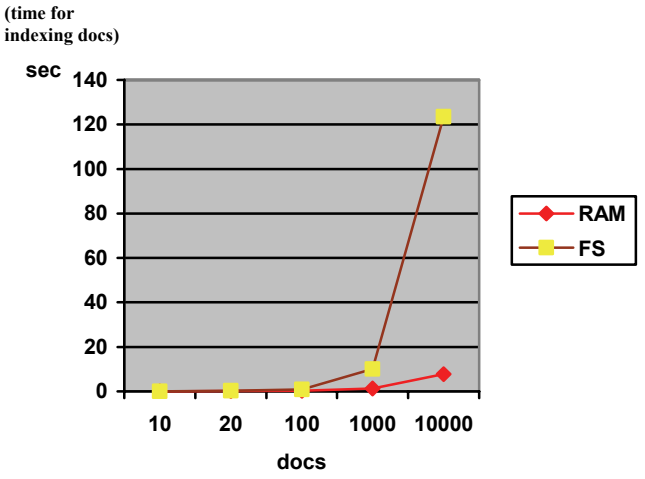
As a result, in CCMS the compound index structure is used. If the indexing times with the multi-file and compound index structures in the search engine application are compared, the results show that the difference between the times is not too much. Also, the compound index stores less files than multi-file index in the file system. Furthermore, searching for documents is faster in the compound index as explained in previous paragraphs.

6.2.2. FS versus RAM Directory

FSDirectory class provides the storage path for a Lucene index which resides in a file system. RAMDirectory class holds an index in memory. Here the performance difference of two different directory types is shown. It is obvious that RAMDirectory is faster than FSDirectory, because in RAMDirectory the index is in memory and this provides faster indexing and searching times.

In FSDirectory, the indexer or searcher program needs to access to the computer disk for writing to the index or reading from index. Therefore, if a user has small size of indexes then probably RAMDirectory would be efficient. But, at the end of implementation one can need to store the indexes on a permanent storage like FSDirectory for further usage, because the index in RAMDirectory is erased after the program termination. These features should be considered in design and implementation of search engine applications.

The following figure 6.2 shows the performance test results of FSDirectory and RAMDirectory indexes:



#docs	RAM (sec.)	FS (sec.)
10	0,050	0,050
20	0,125	0,219
100	0,282	0,890
1000	1,3	10
10000	7,75	123,5

Figure 6.2: RAMDirectory vs. FSDirectory

The test results show that with RAMDirectory the indexing times increase almost linearly, but with FSDirectory while the number of indexed documents increases the time for indexing those goes up faster. The reason for that is number of disk accesses for writing terms to the index consumes more time. Therefore, RAMDirectory is the optimum solution for indexing and searching documents in search engine applications.

On the other hand, both of the Directory types can be used in one application in order to index faster and store it in a file system (if one uses only RAMDirectory, created index is deleted when the application stops). For this case, firstly create an index using RAMDirectory and add all retrieved documents to it. Then, copy the completed index from RAM to FSDirectory. This is the best solution for batch indexing with higher indexing performance.

6.2.3. Index Tuning

The performance analysis applied to Lucene information retrieval library shows that Lucene is able to index documents very fast (according to [Su02], 100 documents per second). On the other hand, Lucene's IndexWriter class has special parameters for tuning the index process. They control Lucene buffer size in memory, segment size and merging frequency during indexing. The parameters are;

1) Merge Factor (mergeFactor – default value is 10)
2) Max Merge Documents (maxMergeDocs)
3) Min Merge Documents (minMergeDocs – default value is 10)

The merge factor determines how often the segment indices will be merged during adding documents to the index. Smaller values use less memory and merge operations are more frequent. Therefore, small values are suitable for interactive indexing and computer systems that have limited memory, whereas larger merge factor values (>10) are optimum for batch indexing but use more memory.

The parameter of maxMergeDocs restricts the number of documents per segment. It works similar to mergeFactor, smaller values are best for interactive indexing (for example, smaller than 10,000) and larger values are best for batch indexing. Also, indexing is faster in larger values.

Finally, the parameter minMergeDocs determines the buffer size in memory for creating documents as segments and later merging them. This directly affects indexing performance and larger values provide faster indexing. In general, the default values works well while indexing. However, if one changes these indexing parameters, he should be careful not having out of memory errors. This can cause index corruption and bad results.

Chapter 7

7. Conclusions

The investigation about information retrieval services for CCMS showed that it was feasible to develop and integrate a full-text search engine application into the CCMS. IR services add tools and many facilities for indexing documents and searching them from the systems easily. Therefore, the implemented search engine application provides efficient methods in order to index assets (concept and content) and retrieve them by user queries. The develop document parsers work well without errors.

The full-text search engine is a complete and useful application that realizes all features that a search engine must have. Furthermore, integration of this Lucene search engine application with Conceptual Content Management System (CCMS) worked very well. The searching and reading retrieved assets from CCMS are executed correctly and fast as planned.

7.1. Future Work

In this part further suggestions and what features of information retrieval (IR) application can be improved are explained. Firstly, the implemented search engine only deals with textual documents which are stored in CCMS's database as an XML data. The textual data are produced by extracting text values from the asset information. But, in other content management systems the document types may differ. For example, the file types could be PDF, HTML, MS Word, etc. For such type of contents, the IR application in CCMS needs specific document parsers for each type.

The fundamental feature of CCMS is support for multi-media content. It can be image, audio or video. Until now, in CCMS assets have 'content' parts and they consist of stored image files. If the contents are replaced by audio or video files, there should be an

improvement in order to retrieve and index information from these files. This will advance the full-text search engine support for any type of data.

As explained in analyzer chapter, there are different analyzer types for different languages in information retrieval libraries. If the documents are written in German, then the program must use a German analyzer or for English content, English analyzer is needed. For this, reason a global analyzer can be implemented that handles most languages or all of them. This will support using many languages in the same content management system. Nowadays, the greatest number of search engines is based on only one language. It is obvious that multi-language support will provide more efficient search engines without considering the language of contents.

On the other hand, as stated there are some drawbacks in stemming especially in German grammar. A better stemming algorithm can be developed for a better content analyzing (also new versions of IR libraries improve the analyzers). An improvement in stemming process means that indexed assets in CCMS will provide more accurate and relevant search results for the users.

As a result the available IR libraries and services are new and under development, but they provide effective and powerful methods in order to develop a full-text search engine that fulfil most of the user requirements and searching features.

REFERENCES:

[ACF] Asset Compiler Framework and Generator Development Guide, Link: http://www.sts.tu-harburg.de/~hw.sehring/cocoma/projs/compiler /Compiler_Framework.pdf

[BR99] Modern Information Retrieval, by Ricardo Baeza-Yates, Berthier Ribeiro-Neto, Addison Wesley 1999, Link: http://www.sims.berkeley.edu/~hearst/irbook/

[Cau99] Jörg Caumanns, A Fast and Simple Stemming Algorithm for German Words, Freie Universität Berlin, October 1999. Link: ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-99-16.ps.gz

[CCM] Open Dynamic Conceptual Content Management, Link: http://www.sts.tu-harburg.de/~hw.sehring/cocoma/

[EGO] Egothor search engine library, Link: http://www.egothor.org/

[eXist] eXist, an Open Source native XML database, Link: http://exist.sourceforge.net/

[FB92] Information Retrieval: Data Structures and Algorithms by William B. Frakes, Ricardo Baeza-Yates, Prentice Hall PTR; Facsimile edition (June 12, 1992)

[GS] Levenshtein Distance, in Three Flavors by Michael Gilleland, Merriam Park Software, Link: http://www.merriampark.com/ld.htm

[Hof99] Probabilistic latent semantic indexing, Thomas Hofmann, Proc. of the 22nd Annual ACM Conference on Research and Development in Information Retrieval Link: http://citeseer.ist.psu.edu/394759.html

[JCD] Jakarta Commons Digester, Open source XML file processing library Link: http://jakarta.apache.org/commons/digester/

[JTi] JTidy, Java HTML processing library, Link: http://jtidy.sourceforge.net/

[LCS97] Document Ranking and the Vector-Space Model, Dik L. Lee, Huei Chuang, Kent Seamons, Link: http://www.cs.ust.hk/faculty/dlee/Papers/ir/ieee-sw-rank.pdf

[LIA04] Lucene in Action by Erik Hatcher, Manning Publications (December 31, 2004), Link: http://www.lucenebook.com

[LUC] Lucene search engine library, Link: http://jakarta.apache.org/lucene

[LUC2] Lucene Index File Formats, http://lucene.apache.org/java/docs/fileformats.html

[LUC3] Lucene 14.3 API, Link: http://lucene.apache.org/java/docs/api/index.html

[PDF] PDFBox, Java PDF processing Library, Link: http://www.pdfbox.org/

[POI] Jakarta POI, Java API to Access Microsoft Format Files, Link: http://jakarta.apache.org/poi/

[Rij] Information Retrieval, A book by C. J. van RIJSBERGEN, Information Retrieval Group, University of Glasgow, Link: http://www.dcs.gla.ac.uk/Keith/Preface.html

[Rob03] "So, what is a content management system?" by James Robertson, Link: http://www.steptwo.com.au/papers/kmc_what/index.html

[Seh04] Hans-Werner Sehring: Konzeptorientierte Inhaltsverwaltung: Modell, Systemarchitektur und Prototypen. Doctoral thesis, Technische Universtität Hamburg-Harburg, 2004, Link: http://www.sts.tu-harburg.de/~hw.sehring/publ/Hans-Werner_Sehring_-_COCoMa.pdf

[SS03] Joachim W. Schmidt and Hans-Werner Sehring: Conceptual Content Modeling and Management: The Rationale of an Asset Language. Proc. PSI'03, 2003, Link: http://www.sts.tu-harburg.de/~hw.sehring/cocoma/publ/2003-PSI03-JWSHWS.pdf

[SS04] Hans-Werner Sehring and Joachim W. Schmidt: Beyond Databases: An Asset Language for Conceptual Content Management. Proc. ADBIS 2004, 2004, Link: http://www.sts.tu-harburg.de/~hw.sehring/cocoma/publ/2004-ADBIS2004-JWSHWS.pdf

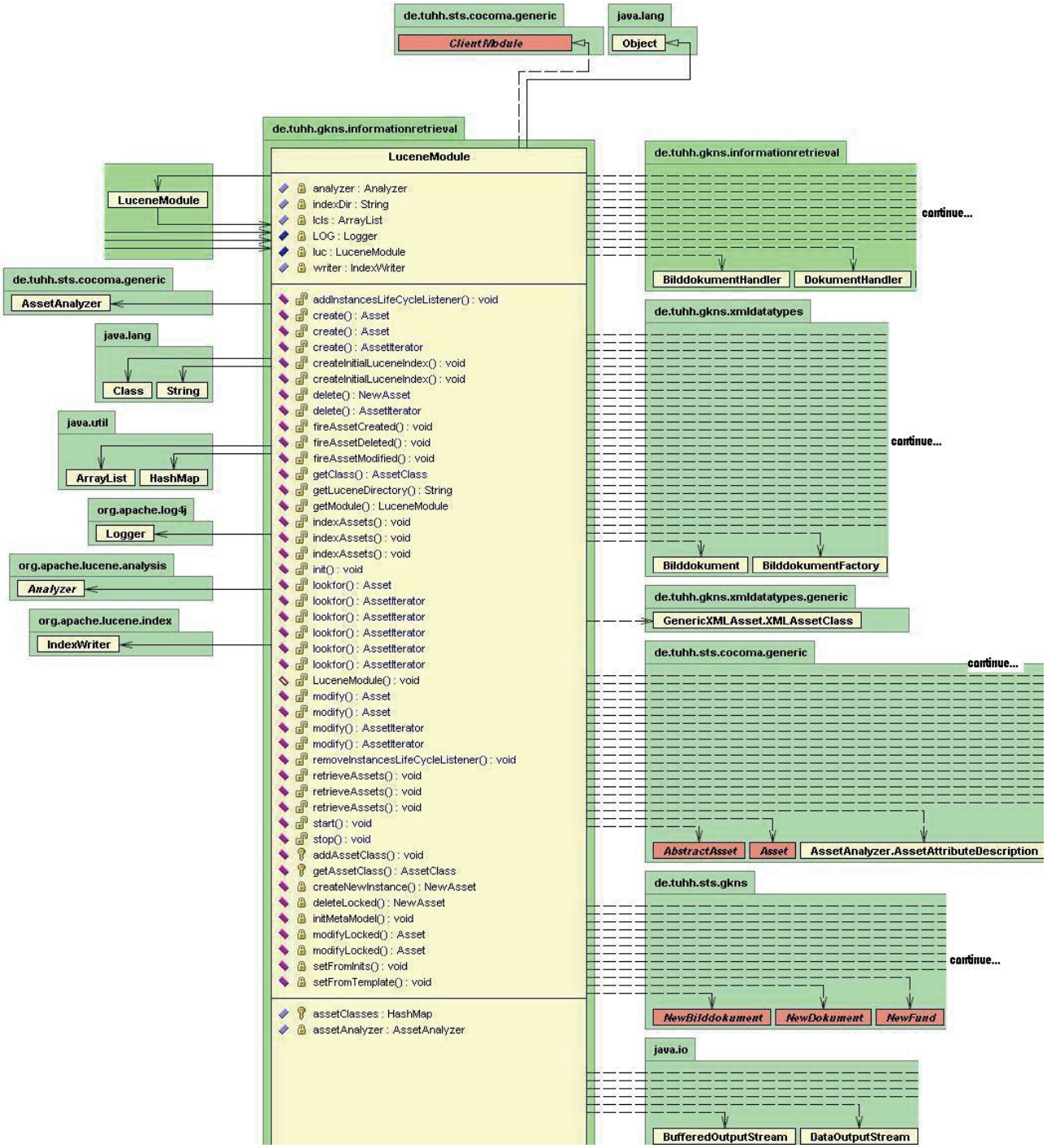
[Su02] Performance Analysis and Optimization on Lucene, David Chi-Chuan Su, Link:http://www.stanford.edu/class/archive/cs/cs276a/cs276a.1032 /projects/reports/dsu800.pdf

[Welib] The Warburg Electronic Library, Link: http://www.welib.de/

[XAP] Xapian, an Open Source Probabilistic Information Retrieval library, Link: http://www.xapian.org/

[XQu] XML Query (XQuery), Link: http://www.w3.org/TR/xquery

APPENDIX A: UML Diagram of Lucene Module



APPENDIX B:

This section shows the important parts of programming codes for Lucene module, search logic of the full-text search engine and one of the document handlers that parses retrieved assets. The full application and source codes are available in a separate CD.

Lucene Module Class:

```
package de.tuhh.gkns.informationretrieval;
.....................

public class LuceneModule implements ClientModule {

public void createInitialLuceneIndex() {

       boolean createFlag= true;
       indexDir = "indexableXMLFiles\\index";  // default index directory
        analyzer = new GermanAnalyzer();

       // IndexWriter to use for adding assets to the index
        try {
              writer = new IndexWriter(indexDir, analyzer, createFlag);
              writer.close();
              } catch (IOException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              }
       }

public void createInitialLuceneIndex(String setDir) {
       IndexWriter writer;
       boolean createFlag= true;
       indexDir = setDir;
        Analyzer analyzer = new GermanAnalyzer();

       // IndexWriter to use for adding assets to the index
        try {
              writer = new IndexWriter(indexDir, analyzer, createFlag);
              writer.close();
       } catch (IOException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
              }
       }

       // indexes all default asset classes
       public void indexAssets() throws IOException, SAXException {
              KorrespondenzHandler kh = new KorrespondenzHandler();
              BilddokumentHandler bh = new BilddokumentHandler();
              DokumentHandler dh = new DokumentHandler();
              FundHandler fh = new FundHandler();
              GesetzerlassbestimmungHandler gbh = new
              GesetzerlassbestimmungHandler();
              LebensdokumentHandler ldh = new LebensdokumentHandler();
              ManuskriptHandler mh = new ManuskriptHandler();
              NachlassHandler nh = new NachlassHandler();
              SachakteHandler sh = new SachakteHandler();
              VeroeffentlichungHandler vh = new VeroeffentlichungHandler();
              UserHandler uh = new UserHandler();
```

```
            // here new documents are appended to the existing index
            kh.execute(indexDir);
            bh.execute(indexDir);
            dh.execute(indexDir);
            fh.execute(indexDir);
            gbh.execute(indexDir);
            ldh.execute(indexDir);
            mh.execute(indexDir);
            nh.execute(indexDir);
            sh.execute(indexDir);
            vh.execute(indexDir);
            uh.execute(indexDir);

            // Finally, optimize the index and close
            writer = new IndexWriter(indexDir, analyzer, false);
            writer.optimize();
            writer.close();
      }

      public void indexAssets(AssetClass asset) throws IOException,
SAXException {

      // The code for indexing a specific type of asset class
      }

      public void indexAssets(AssetClass[] asset) throws IOException,
SAXException {

      // The code for indexing multiple types of asset classes
}
.......................

      /*
       * with default Queries retrieve all assets from the database
       * The different type of assets must be retrieved with their special
queries.
       * Because, each asset model has their own XML schema or structure and
       * there are specific parsers for each of them in order to parse XML
stream.
      */

public void retrieveAssets() {
      String queryKorres = "declare namespace
      gkns='http://sts.tuhh.de/gkns/dokumenttypen.xsd'; " +
      "<gkns:allAssetList>{ /child::gkns:*[local-name(.)='korrespondenz' ]}" +
      "</gkns:allAssetList>";

      String queryFund = "declare namespace
      gkns='http://sts.tuhh.de/gkns/dokumenttypen.xsd'; " +
      "<gkns:allAssetList>{ /child::gkns:*[local-name(.)='fund' ]}" +
      "</gkns:allAssetList>";

      String queryDok = "declare namespace
      gkns='http://sts.tuhh.de/gkns/dokumenttypen.xsd'; " +
      "<gkns:allAssetList>{ /child::gkns:*[local-name(.)='dokument' ]}" +
      "</gkns:allAssetList>";

      .....................
      // same as for other asset types
      .....................

      // for Korrespondenz asset type
      AssetIterator lookIt;
      lookIt = luc.lookfor(query);
      System.out.println(lookIt.getLength());
      lookIt = luc.lookfor(queryKorres);
```

```
	try {
		DataOutputStream out = new DataOutputStream (
		new BufferedOutputStream(
		new FileOutputStream("indexableXMLFiles\\korrespondenz.xml")));

		out.writeBytes("<?xml version=\"1.0\" encoding=\"windows-
1252\"?>");
		out.writeBytes("<data>");
		while(lookIt.hasNext())
		{
			String data = lookIt.next().toString();
			System.out.println(data);
			out.writeBytes(data);
		}
		out.writeBytes("</data>");
		out.close();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		// for Fund asset type
		lookIt = luc.lookfor(queryFund);
		try {
			DataOutputStream out = new DataOutputStream (
			new BufferedOutputStream(
			new FileOutputStream("indexableXMLFiles\\fund.xml")));

			out.writeBytes("<?xml version=\"1.0\" encoding=\"windows-
1252\"?>");
			out.writeBytes("<data>");
			while(lookIt.hasNext())
			{
				String data = lookIt.next().toString();
				System.out.println(data);
				out.writeBytes(data);
			}
			out.writeBytes("</data>");
			out.close();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		// for Dokument asset type
		lookIt = luc.lookfor(queryDok);
		try {
			DataOutputStream out = new DataOutputStream (
			new BufferedOutputStream(
			new FileOutputStream("indexableXMLFiles\\dokument.xml")));

			out.writeBytes("<?xml version=\"1.0\" encoding=\"windows-
1252\"?>");
			out.writeBytes("<data>");
			while(lookIt.hasNext())
			{
				String data = lookIt.next().toString();
				System.out.println(data);
				out.writeBytes(data);
			}
			out.writeBytes("</data>");
			out.close();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
```

```
        ......................
        // same as for other asset types
        ......................
        }

        public void retrieveAssets(AssetClass asset, String xquery) {

        // with this method user can retrieve a specific type of asset with a
query
        // example: retrieve(Lebensdokument, xquery)
        }

        public void retrieveAssets(AssetClass[] asset, String[] xquery) {

        // with this method user can retrieve multiple types of assets with
        corresponding asset queries

        }

        // returns the used LUCENEModule
        public LuceneModule getModule() {
               return luc;
        }

        // returns the index directory
        public String getLuceneDirectory() {
               return indexDir;
        }
}
```

Document Parser (DokumentHandler class):

```
package de.tuhh.gkns.informationretrieval;
.......................


/**
 * Parses the contents of Dokument XML stream and indexes all
 * asset entries found in it.
 */
public class DokumentHandler
{
        private static IndexWriter fsWriter;

        /**
      * Adds the Dokument asset to the index.
      *
      * @param asset in order to add to the index
      */

     public void addDokumentAsset(Dokument asset) throws IOException
     {
        System.out.println("Adding " + asset.getId());
        Document assetDocument  = new Document();
        assetDocument.add(Field.Keyword("id", asset.getId()));
        assetDocument.add(Field.Text("typ", asset.getTyp()));
        assetDocument.add(Field.Keyword("datum", asset.getDatum()));
        assetDocument.add(Field.Text("titel", asset.getTitel()));
        assetDocument.add(Field.Keyword("erfassungsdatum",
        asset.getErfassungsdatum()));
        assetDocument.add(Field.Text("bemerkung", asset.getBemerkung()));
        assetDocument.add(Field.Keyword("erfasserRef",
        asset.getErfasserRef()));
```

```
      assetDocument.add(Field.Keyword("contentIds", asset.getContentIds()));
      assetDocument.add(Field.Text("entstehungsort",
      asset.getEntstehungsort()));
      assetDocument.add(Field.Text("inhalt", asset.getInhalt()));
      assetDocument.add(Field.Text("umfang", asset.getUmfang()));

   assetDocument.add(Field.UnStored("contents", asset.getId() + " " +
   asset.getTyp() + " " +asset.getTitel() + " " + asset.getBemerkung() + " " +
   asset.getErfasserRef() + " " + asset.getEntstehungsort() + " " +
   asset.getInhalt() + " " + asset.getUmfang() + " " +
    asset.getContentIds()));

      fsWriter.addDocument(assetDocument);
      System.out.println(assetDocument);
   }

   /**
    * Refers to the index to add assets to, configures Digester rules and
    * actions, parses the Dokument XML file.
    *
    * @param Directory where the index is created
    */
   public void execute(String setDir) throws IOException, SAXException
   {
      String indexDir = setDir;
      Analyzer analyzer = new GermanAnalyzer();
      boolean createFlag = false;
      // this flag means append to existing index without recreating

      // IndexWriter to use for adding assets to the index
      fsWriter = new IndexWriter(indexDir, analyzer, createFlag);

      // instantiate Digester and disable XML validation
      Digester digester = new Digester();
      digester.setValidating(false);

      // instantiate DokumentHandler class
      digester.addObjectCreate("data", DokumentHandler.class );

      // instantiate asset class
      digester.addObjectCreate("data/xml-fragment", Dokument.class );

      // set id property of asset instance when 'id' attribute is found
      digester.addSetProperties("data/xml-fragment","id", "id" );

      // set different properties of asset instance using specified methods
      digester.addCallMethod("data/xml-fragment/dok:typ", "setTyp", 0);
      digester.addCallMethod("data/xml-fragment/dok:datum", "setDatum", 0);
      digester.addCallMethod("data/xml-fragment/dok:titel", "setTitel", 0);
      digester.addCallMethod("data/xml-fragment/dok:erfassungsdatum",
      "setErfassungsdatum", 0);
      digester.addCallMethod("data/xml-fragment/dok:bemerkung",
"setBemerkung", 0);
      digester.addCallMethod("data/xml-fragment/dok:erfasserRef",
"setErfasserRef", 0);
      digester.addCallMethod("data/xml-fragment/dok:contentIds",
"setContentIds", 0);
      digester.addCallMethod ( "data/xml-fragment/dok:entstehungsort",
      "setEntstehungsort", 0);
      digester.addCallMethod("data/xml-fragment/dok:inhalt", "setInhalt", 0);
      digester.addCallMethod("data/xml-fragment/dok:umfang", "setUmfang", 0);

      // call 'addDokumentAsset' method when the next 'xml-fragment' pattern
      // is seen
      digester.addSetNext("data/xml-fragment", "addDokumentAsset" );
```

```
        // now that rules and actions are configured, start the parsing process
        InputSource is = new InputSource(new
        FileInputStream("indexableXMLFiles\\dokument.xml"));

        DokumentHandler dml = (DokumentHandler)digester.parse(is);
        fsWriter.close();
    }

.....................
    /**
     * JavaBean class that holds properties of each asset entry.
     * It is important that this class be public and static, in order for
     * Digester to be able to instantiate it.
     */
    public static class Dokument{
       private String id;
       private String typ;
       private String datum;
       private String titel;
       private String erfassungsdatum;
       private String bemerkung;
       private String erfasserRef;
       private String entstehungsort;
       private String inhalt;
       private String umfang;
       private String contentIds;

       public void setContentIds(String newContentIds)
       {
              contentIds = newContentIds;
        }
       public String getContentIds()
        {
              return contentIds;
        }

       public void setId(String newId)
       {
              id = newId;
        }
       public String getId()
       {
              return id;
       }

        public void setTyp(String newTyp)
        {
              typ = newTyp;
        }
        public String getTyp()
        {
              return typ;
        }

       public void setDatum(String newDatum)
       {
              datum = formatDate(newDatum);
       }
       public String getDatum()
       {
              return datum;
       }
.....................
       // same as for other Dokument JavaBean class properties.
.....................

}
```

Search Logic Class:

```
package de.tuhh.gkns.informationretrieval;
.....................
// import libraries
.....................

public class SearchLogic {
	private String input;
	// default index directory, you can change it with method "setDir()"
	private File indexDir = new File("indexableXMLFiles\\index");
	private static Hits hits;
	private static Document doc;
	private static long end;
	private static long start;

	SearchLogic(String inputFromUI) throws Exception
	{
		this.input = inputFromUI;
		search(indexDir, input);
	}

	public static void search(File indexDir, String q) throws Exception {
	    // Refer to the created Lucene index in the directory
	    Directory fsDir = FSDirectory.getDirectory(indexDir, false);
	    IndexSearcher is = new IndexSearcher(fsDir);

	    // parse the query 'q
	    Query query = QueryParser.parse(q, "contents",
	                                    new GermanAnalyzer());
	    start = new Date().getTime();
	    // perform the search operation
	    hits = is.search(query);
	    end = new Date().getTime();

	    System.err.println("Found " + hits.length() +
	      " document(s) (in " + (end - start) +
	      " milliseconds) that matched query '" + q + "':");

	    for (int i = 0; i < hits.length(); i++) {
	      doc = hits.doc(i);
	      System.out.println(doc.get("id"));
	    }
	    is.close();
	    fsDir.close();
	}

	// returns the search hits
	public Hits getDocs() {
		return hits;
	}

	// returns the search time
	public long getSearchTime() {
		return (end-start);
	}

	// set the path of index
	public void setDir(String setDir) {
		indexDir = new File(setDir);
	}

	public File getDir() {
		return indexDir;
	}
}
```

APPENDIX C

The Asset modeling schema [CCMS]:

```
<?xml version='1.0' encoding='utf-8'?>
<xs:schema targetNamespace="http://sts.tuhh.de/gkns/dokumenttypen.xsd"
xmlns:gkns="http://sts.tuhh.de/gkns/dokumenttypen.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xdt="http://www.w3.org/2003/05/xpath-
datatypes" elementFormDefault="qualified">

        <xs:element name="referenz" type="gkns:Referenz"/>
        <xs:element name="allAssetList" type="gkns:AllAssetList"/>
        <xs:element name="muMeDokument" type="gkns:MuMeDokument"/>
        <xs:element name="nachlass" type="gkns:Nachlass"/>
        <xs:element name="ankuendigung" type="gkns:Ankuendigung"/>
        <xs:element name="irrelevanterFund" type="gkns:IrrelevanterFund"/>
        <xs:element name="dokument" type="gkns:Dokument"/>
        <xs:element name="fund" type="gkns:Fund"/>
        <xs:element name="bilddokument" type="gkns:Bilddokument"/>
        <xs:element name="mask" type="gkns:Mask"/>
        <xs:element name="sachakte" type="gkns:Sachakte"/>
        <xs:element name="tondokument" type="gkns:Tondokument"/>
        <xs:element name="lebensdokument" type="gkns:Lebensdokument"/>
        <xs:element name="korrespondenz" type="gkns:Korrespondenz"/>
        <xs:element name="veroeffentlichung" type="gkns:Veroeffentlichung"/>
        <xs:element name="user" type="gkns:User"/>
        <xs:element name="schlagwort" type="gkns:Schlagwort"/>
        <xs:element name="manuskript" type="gkns:Manuskript"/>
        <xs:element name="gesetzErlassBestimmung" type="gkns:GesetzErlassBestimmung"/>
        <xs:element name="kommentar" type="gkns:Kommentar"/>

        <xs:complexType name="Referenz">
                <xs:sequence>
                        <xs:element name="signatur" type="xs:string"/>
                        <xs:element name="ort" type="xs:string"/>
                </xs:sequence>
                <xs:attribute name="id" type="xs:ID" use="required"/>
        </xs:complexType>
        <xs:complexType name="IrrelevanterFund">
                <xs:complexContent>
                        <xs:extension base="gkns:Fund">
                                <xs:sequence></xs:sequence>
                        </xs:extension>
                </xs:complexContent>
        </xs:complexType>

        <xs:complexType name="MuMeDokument">
                <xs:complexContent>
                        <xs:extension base="gkns:Dokument">
                        <xs:sequence>
                        <xs:element name="autor" type="xs:string"/>
                        <xs:element name="beteiligtePersonen" type="xs:string"/>
                        </xs:sequence>
                        </xs:extension>
                </xs:complexContent>
        </xs:complexType>
```

```
<xs:complexType name="Nachlass">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
		<xs:sequence>
		<xs:element name="nachlasser" type="xs:string"/>
		</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Ankuendigung">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
			<xs:sequence>
				<xs:element name="anlass" type="xs:string"/>
				<xs:element name="ort" type="xs:string"/>
			</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Fund">
	<xs:sequence>
		<xs:element name="typ" type="xs:string"/>
		<xs:element name="datum" type="xs:dateTime"/>
		<xs:element name="titel" type="xs:string"/>
		<xs:element name="erfassungsdatum" type="xs:dateTime"/>
		<xs:element name="bemerkung" type="xs:string"/>
		<xs:element ref="gkns:kommentar" minOccurs="0"
		maxOccurs="unbounded"/>
		<xs:element ref="gkns:mask" minOccurs="0" maxOccurs="unbounded"/>
		<xs:element name="erfasserRef" type="xs:ID" minOccurs="0"/>
		<xs:element ref="gkns:referenz" minOccurs="0"/>
		<xs:element name="verschlagwortungRef" type="xs:ID" minOccurs="0"
		maxOccurs="unbounded"/>
		<xs:element name="contentIds" type="xs:string"/>
	</xs:sequence>
	<xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>


<xs:complexType name="Dokument">
	<xs:complexContent>
		<xs:extension base="gkns:Fund">
		<xs:sequence>
		<xs:element name="entstehungsort" type="xs:string"/>
		<xs:element name="sperrvermerkFachlich" type="xs:dateTime"/>
		<xs:element name="sperrvermerkJuristisch" type="xs:dateTime"/>
		<xs:element name="inhalt" type="xs:string"/>
		<xs:element name="umfang" type="xs:string"/>
		</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="AllAssetList">
	<xs:sequence>
		<xs:choice minOccurs="0" maxOccurs="unbounded">
			<xs:element ref="gkns:referenz"/>
			<xs:element ref="gkns:muMeDokument"/>
			<xs:element ref="gkns:nachlass"/>
			<xs:element ref="gkns:ankuendigung"/>
			<xs:element ref="gkns:irrelevanterFund"/>
			<xs:element ref="gkns:dokument"/>
			<xs:element ref="gkns:fund"/>
			<xs:element ref="gkns:bilddokument"/>
			<xs:element ref="gkns:mask"/>
			<xs:element ref="gkns:sachakte"/>
			<xs:element ref="gkns:tondokument"/>
			<xs:element ref="gkns:lebensdokument"/>
			<xs:element ref="gkns:korrespondenz"/>
			<xs:element ref="gkns:veroeffentlichung"/>
			<xs:element ref="gkns:user"/>
			<xs:element ref="gkns:schlagwort"/>
			<xs:element ref="gkns:manuskript"/>
			<xs:element ref="gkns:gesetzErlassBestimmung"/>
			<xs:element ref="gkns:kommentar"/>
		</xs:choice>
	</xs:sequence>
</xs:complexType>


<xs:complexType name="Bilddokument">
	<xs:complexContent>
		<xs:extension base="gkns:MuMeDokument">
			<xs:sequence></xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Mask">
	<xs:sequence>
		<xs:element name="w" type="xs:integer"/>
		<xs:element name="h" type="xs:integer"/>
		<xs:element name="y" type="xs:integer"/>
		<xs:element name="contentId" type="xs:string"/>
		<xs:element name="x" type="xs:integer"/>
	</xs:sequence>
	<xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>

<xs:complexType name="Sachakte">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
		<xs:sequence>
		<xs:element name="betroffeneInstitution" type="xs:string"/>
		<xs:element name="betroffener" type="xs:string"/>
		<xs:element name="inhalt" type="xs:string"/>
		</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="Tondokument">
	<xs:complexContent>
		<xs:extension base="gkns:MuMeDokument">
			<xs:sequence></xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Lebensdokument">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
		<xs:sequence>
		<xs:element name="person" type="xs:string"/>
		<xs:element name="aussteller" type="xs:string"/>
		</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Korrespondenz">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
		<xs:sequence>
		<xs:element name="adressatInstitution" type="xs:string"/>
		<xs:element name="absenderInstitution" type="xs:string"/>
		<xs:element name="betreff" type="xs:string"/>
		<xs:element name="adressat" type="xs:string"/>
		<xs:element name="absender" type="xs:string"/>
		</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="Veroeffentlichung">
	<xs:complexContent>
		<xs:extension base="gkns:Dokument">
			<xs:sequence>
				<xs:element name="autor" type="xs:string"/>
				<xs:element name="jahr" type="xs:integer"/>
			</xs:sequence>
		</xs:extension>
	</xs:complexContent>
</xs:complexType>


<xs:complexType name="User">
	<xs:sequence>
		<xs:element name="name" type="xs:string"/>
		<xs:element name="kurz" type="xs:string"/>
	</xs:sequence>
	<xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
```

```
	<xs:complexType name="Schlagwort">
		<xs:sequence>
			<xs:element name="name" type="xs:string"/>
			<xs:element name="identifier" type="xs:integer"/>
			<xs:element name="kommentar" type="xs:string" minOccurs="0"/>
			<xs:element name="kuerzel" type="xs:string" minOccurs="0"/>
			<xs:element ref="gkns:schlagwort" minOccurs="0"
			maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="id" type="xs:ID" use="required"/>
	</xs:complexType>


	<xs:complexType name="Manuskript">
		<xs:complexContent>
			<xs:extension base="gkns:Dokument">
				<xs:sequence>
					<xs:element name="autor" type="xs:string"/>
				</xs:sequence>
			</xs:extension>
		</xs:complexContent>
	</xs:complexType>


	<xs:complexType name="GesetzErlassBestimmung">
		<xs:complexContent>
			<xs:extension base="gkns:Dokument">
			<xs:sequence>
			<xs:element name="erlasser" type="xs:string"/>
			<xs:element name="wirksamkeit" type="xs:dateTime"/>
			<xs:element name="betreff" type="xs:string"/>
			<xs:element name="adressat" type="xs:string"/>
			<xs:element name="voeDatum" type="xs:dateTime"/>
			<xs:element name="unterzeichner" type="xs:string"/>
			</xs:sequence>
			</xs:extension>
		</xs:complexContent>
	</xs:complexType>


	<xs:complexType name="Kommentar">
		<xs:sequence>
			<xs:element name="datum" type="xs:dateTime"/>
			<xs:element name="text" type="xs:string"/>
			<xs:element name="autorRef" type="xs:ID" minOccurs="0"/>
			<xs:element name="antwortenRef" type="xs:ID" minOccurs="0"
			maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="id" type="xs:ID" use="required"/>
	</xs:complexType>

</xs:schema>
```