Diplomarbeit

Entwicklung einer Suchmaschine für wissenschaftliche Dokumente

Peter Pakula
Fachhochschule Köln

DLR

Fachhochschule Köln
Cologne University of Applied Sciences

Campus Gummersbach

Entwicklung einer Suchmaschine für wissenschaftliche Dokumente

Diplomarbeit

vorgelegt an der Fachhochschule Köln
Campus Gummersbach

im Studiengang Wirtschaftsinformatik

ausgearbeitet von :
Peter Pakula
geboren am 24.08.1977
Matr.-Nr.: 11041557

Erstprüfer: Prof. Dr. Heide Faeskorn-Woyke
Zweitprüfer: Prof. Dr. Frank Victor
Fachhochschule Köln Campus Gummersbach

Betreuer: Dipl.-Math. Jens Rühmkorf
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Köln

Gummersbach, im April 2009

Danksagung

Ich möchte meinen Eltern, Janina und Henryk Pakula sowie meiner wundervollen Freundin Karoline Adamczyk für ihre Unterstützung danken.

Ebenso danke ich Jens Rühmkorf, meinen Betreuer beim DLR und Frau Prof. Dr. Heide Faeskorn-Woyke für die Betreuung.

Herrn Prof. Dr. Frank Victor danke ich für die Übernahme der Zweitprüfung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Entwicklung von Luft- und Raumfahrt Fahrzeugen ist von hoher Komplexität geprägt und erfordert in besonderem Maße das Zusammenspiel verschiedenster Disziplinen. In großen Projekten, wie etwa dem Flugzeugentwurf, werden Komponenten von Instituten unterschiedlicher Bereiche wie z.B. Aerodynamik und Strömungstechnik, Antriebstechnik, Werkstoff-Forschung sowie Hochfrequenztechnik entwickelt.

Hohe Investitionen und lange Entwicklungszyklen in der Luft- und Raumfahrt erfordern eine gewisse Sensibilität beim Umgang mit digitalem Wissen aus den beteiligten Bereichen. Zwar ist es in der Luft- und Raumfahrtforschung genauso wie in anderen Forschungsgebieten unverzichtbar, Neuerungen regelmäßig zu veröffentlichen und in der Fachwelt zur Diskussion zu stellen, hohe internationale Konkurrenz verhindert jedoch, dass unternehmensinterne Dokumentationen unbeschränkt zur Verfügung gestellt werden können.

Um es Wissenschaftlern zu erleichtern, sich in andere Bereiche einzuarbeiten und eine schnelle Bereitstellung geeigneter Werkzeuge zu gewährleisten, wird im Rahmen eines internen Projekts des Deutschen Zentrums für Luft- und Raumfahrt ein Expertensystem zur Anwenderunterstützung bei flugphysikalischen Simulationen entwickelt. Dieses Werkzeug soll es Wissenschaftlern erleichtern, sich in andere Bereiche und die Verwendung ihrer Werkzeuge einzuarbeiten. Das zu entwickelnde System soll Expertenwissen dem Anwender sowohl strukturiert als auch unstrukturiert zur Verfügung stellen. Der unstrukturierte Zugriff auf Expertenwissen soll über eine Volltextsuchmaschine realisiert werden, die es einem Wissenschaftler ermöglicht, sich relevante Literatur eines Fachgebietes zügig zu erschließen. Strukturiertes Wissen wird in das System in Form von Regeln und Prozessen durch einen Experten eingepflegt. Wissenschaftliche Dokumente werden in Form von gängigen Dokumenttypen wie Word-, Excel-, Powerpoint- und PDF-Dateien in das System integriert.

1.2 Das Deutsche Zentrum für Luft- und Raumfahrt im Überblick

Das Deutsche Zentrum für Luft- und Raumfahrt (DLR) ist das Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt, Energie und Verkehr. Über die eigene Forschung hinaus ist das DLR als Raumfahrtagentur im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig.

Das DLR beschäftigt an seinen 29 Instituten bzw. Test- und Betriebseinrichtungen ca. 6.000 Mitarbeiterinnen und Mitarbeiter und ist an den 13 Standorten Köln-Porz, Berlin-Adlershof, Bonn-Oberkassel, Braunschweig, Bremen, Göttingen, Hamburg, Lampoldshausen, Neustrelitz, Oberpfaffenhofen, Stuttgart, Trauen und Weilheim vertreten. Außerdem unterhält das DLR Außenbüros in Brüssel, Paris und Washington D.C.

Die Einrichtung Simulations- und Softwaretechnik (SISTEC) ist auf die DLR-Standorte Köln-Porz und Braunschweig verteilt und gliedert sich in die folgenden Abteilungen:

- Verteilte Systeme und Komponentensoftware und
- Software Qualitätssicherung und eingebettete Systeme

Diese Abteilungen stellen gleichzeitig die derzeitigen Themenschwerpunkte dar. Die Simulations- und Softwaretechnik hat ihre Kompetenz im Bereich des Software-Engineerings.

1.3 Zielsetzung

Ziel dieser Diplomarbeit ist es, eine Volltextsuchmaschine für wissenschaftliche Dokumente zu entwickeln. Diese Suchmaschine soll modular konzipiert sein und es ermöglichen, in andere Programme und Bibliotheken eingebunden zu werden. Gleichzeitig soll ihre Funktionalität als Webservice innerhalb eines Firmen-Intranets angesprochen werden können.

1.4 Gliederung der Diplomarbeit

Die vorliegende Diplomarbeit wurde in sieben Kapitel aufgeteilt. Zunächst werden in Kapitel 2 die Grundlagen des Information Retrieval vorgestellt. Diese geben eine Einführung in die Funktionsweise von Volltextsuchmaschinen. Zu Beginn wird der Begriff Information Retrieval definiert und vom Daten Retrieval abgegrenzt. Anschließend wird eine allgemeine Funktionsweise sowie eine Übersicht der Retrieval-Modelle vorgestellt. Das Boolesche-

sowie das Vektorraum- Modell werden detailliert beschrieben. Zusätzlich wird der gesamte Retrieval-Prozess von Datenaufbereitung, Datenstruktur, Suchanfragen bis hin zu den Suchergebnissen beschrieben.

In Kapitel 3 werden die verwendeten Technologien wie Apache Lucene sowie die Eclipse-Rich-Client-Platform aufgeführt. Dazu werden die relevanten Klassen, die zum Indexieren, Suchen und zur Analyse von Dokumenten der Apache Lucene API genutzt werden, vorgestellt. Anschließend folgt eine Einführung in das OSGi-Framework Eclipse-Rich-Client-Platform und deren Erweiterbarkeit durch Plugins.

Der praktische Teil der Arbeit teilt sich in die Anforderungsanalyse, den Entwurf und die Implementierung des Systems auf. In Kapitel 4 wird die Suchmaschine in einer Gesamtsystemübersicht dargestellt und anschließend von anderen Komponenten abgegrenzt. Es werden funktionale, nicht-funktionale Anforderungen sowie Randbedingungen definiert. Abschließend werden Anwendungsfälle beschrieben.

In dem darauf folgenden Kapitel 5 wird die Software entworfen. Diese teilt sich in drei Softwarekomponenten auf: den Dateisystem-Indexer, die Suchmaske als Webseite im Intranet sowie den Such-RCP-Plugin für das Expertensystem.

Nach dem Entwurf der Softwarekomponenten folgt in Kapitel 6 die detaillierte Beschreibung der Implementierung. Es werden Test-Klassen, Parser sowie die eigentlichen Indexer-Klassen vorgestellt. Zudem folgen noch Klassen der Intranet-Webseite sowie des Such-RCP-Plugins.

Abschließend wird in Kapitel 7 die vorliegende Diplomarbeit zusammengefasst und ein Ausblick gegeben.

Diese Arbeit wurde in der Einrichtung "Simulations- und Softwaretechnik" des Deutschen Zentrums für Luft- und Raumfahrt e.V. verfasst.

2 Grundlagen des Information Retrieval

Thema des „Information Retrieval“ ist die Suche nach Dokumenten. Dabei spielt es keine Rolle, um welche Art von Dokument es sich handelt. Traditionell liegt der Fokus auf Textdokumenten, jedoch stehen heute bei diesem Forschungsgebiet mehr die multimedialen Dokumente wie Bilder, Video und Audio im Mittelpunkt. Durch das Internet hat das Gebiet des Information Retrieval an Bedeutung gewonnen. Einer der Gründe dafür ist die steigende Anzahl der digitalen Dokumente, die im Internet verbreitet werden. Diese sollen möglichst bei der Anfrage eines Anwenders wiedergefunden werden.

Der Begriff Information Retrieval

Der Begriff Information Retrieval teilt sich in die zwei Begriffe „Information“ und „Retrieval“ auf, die hier genauer betrachtet werden. Der Begriff der Information steht in einem Zusammenhang weiterer elementarer Begriffe: Daten und Wissen.

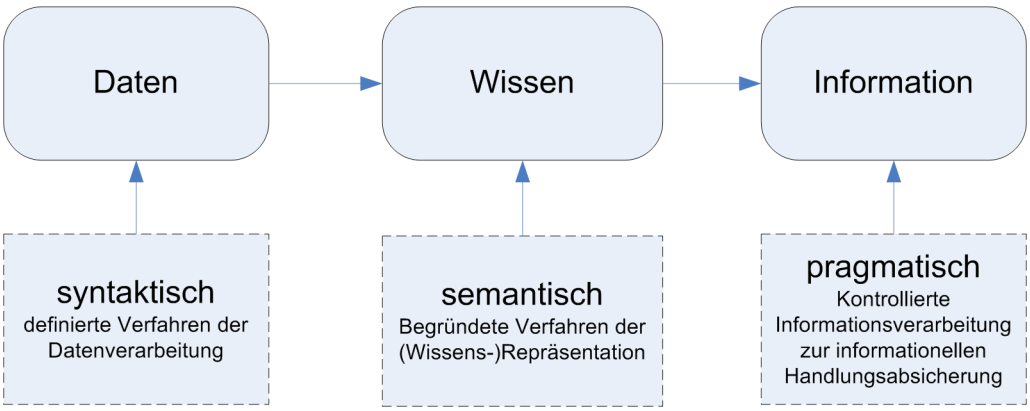

Abbildung 1: Daten, Wissen und Information nach Norbert Fuhr

Daten, Wissen und Information

Abbildung 1 zeigt die Beziehung zwischen Daten, Wissen und Information. Die Daten sind auf der syntaktischen Ebene angesiedelt. Das bedeutet, dass Daten nur eine Sammlung von Werten sind, die keine Struktur aufweisen. Wissen hingegen ist auf der semantischen Ebene angesiedelt und entsteht, wenn Daten strukturiert werden. Laut Rainer Kuhlen [KUH90] *„enthalten also Datenbanksysteme nicht nur Daten, sondern auch Wissen, weil zusätzlich zu den Daten zumindest eine Teil der Semantik des jeweiligen Anwendungsgebietes auch im System modelliert wird“*. Information hingegen befindet sich auf der pragmatischen Ebene.

Damit lässt sich Information nach Rainer Kuhlen [KUH90] wie folgt formulieren: *„Information ist Teilmenge von Wissen, die von jemanden in einer konkreten Situation zur Lösung von Problemen benötigt wird“*. Da Information eine Teilmenge von Wissen ist und in einer konkreten Situation benötigt wird, muss aus einem Informationssystem die benötigte Information extrahiert werden. Als Beispiel dient hier die freie Enzyklopädie Wikipedia. Wenn ein Anwender etwas über das Thema Information Retrieval erfahren möchte, so ist die benötigte Information eine Teilmenge aus der gesamten Menge an Wissen, das Wikipedia zu Verfügung stellt. Dieses Beispiel zeigt auf, dass die Aufgabe eines Information Retrieval-Systems darin besteht, aus einem gespeicherten Informationspool die benötigte Information zu extrahieren und das Informationsbedürfnis des Anwenders zu stillen, aber auch die Information so aufbereiten, dass man darauf zugreifen kann.

Retrieval

Der Begriff Retrieval wird im Fachwörterbuch Computer-Englisch wie folgt übersetzt:

- Wiederfinden
- Wiedergewinnung (von Daten)

Damit kann der Begriff Information Retrieval laut Henrich [HEN08] als *„(Wieder-)Gewinnung von Informationen, die von jemandem in einer konkreten Situation zur Lösung von Problemen benötigt werden“* beschrieben werden.

2.1 Definition und Abgrenzung von Information Retrieval-Systemen

Definition Information Retrieval

Eine klassische Definition des Information Retrieval von Salton und McGill [FUH06] lautet:

„Gegenstand des Information Retrieval ist die Repräsentation, Speicherung und Organisation von Informationen und der Zugriff zu Informationen.“ (Salton/McGill 1987, S.1)

Die Fachgruppe Information Retrieval der Gesellschaft für Informatik [FUH96] beschreibt ihre Ziele und Aufgaben wie folgt:

„Im Information Retrieval (IR) werden Informationssysteme in Bezug auf ihre Rolle im Prozeß des Wissenstransfers vom menschlichen Wissensproduzenten zum Informations-Nachfragenden betrachtet. Die Fachgruppe „Information Retrieval`` in der Gesellschaft für Informatik beschäftigt sich dabei schwerpunktmäßig mit jenen Fragestellungen, die im Zusammenhang mit vagen Anfragen und unsicherem Wissen entstehen. Vage Anfragen sind dadurch gekennzeichnet, dass die Antwort a priori nicht eindeutig definiert ist. Hierzu zählen neben Fragen mit unscharfen Kriterien insbesondere auch solche, die nur im Dialog iterativ durch Reformulierung (in Abhängigkeit von den bisherigen Systemantworten) beantwortet werden können; häufig müssen zudem mehrere Datenbasen zur Beantwortung einer einzelnen Anfrage durchsucht werden.

Die Darstellungsform des in einem IR-System gespeicherten Wissens ist im Prinzip nicht beschränkt (z.B. Texte, multimediale Dokumente, Fakten, Regeln, semantische Netze). Die Unsicherheit (oder die Unvollständigkeit) dieses Wissens resultiert meist aus der begrenzten Repräsentation von dessen Semantik (z.B. bei Texten oder multimedialen Dokumenten); darüber hinaus werden auch solche Anwendungen betrachtet, bei denen die gespeicherten Daten selbst unsicher oder unvollständig sind (wie z.B. bei vielen technisch-wissenschaftlichen Datensammlungen). Aus dieser Problematik ergibt sich die Notwendigkeit zur Bewertung der Qualität der Antworten eines Informationssystems, wobei in einem weiteren Sinne die Effektivität des Systems in Bezug auf die Unterstützung des Benutzers bei der Lösung seines Anwendungsproblems beurteilt werden sollte.“

Diese Definition ist sehr allgemein gehalten und betont besonders die Faktoren vager Anfrage, die Unsicherheit des gespeicherten Wissens und unscharfe Kriterien. Eine vage Anfrage resultiert aus einer diffusen Formulierung des Anwenders. Dadurch enthält ein Information Retrieval-System vage Bedingungen, die keine exakten Antworten liefern. Die Unsicherheit entsteht durch den Mangel eines Systems, tatsächliche Inhalte eines Dokuments anzuzeigen. Um die Charakteristika von Information Retrieval besser zu erläutern wird im Folgenden eine Abgrenzung zwischen Daten und Information Retrieval gemacht.

Abgrenzung Daten Retrieval und Information Retrieval

Die Tabelle 1 orientiert sich an Van Rijsbergens und stellt stichwortartig die Unterschiede zwischen Daten Retrieval und Information Retrieval auf.

	Daten Retrieval	**Information Retrieval**
Matching	Exact Match	Partial match, Best match
Inference	Deduction	Induction
Model	Deterministic	Probabilistic
Classifikation	Monothetic	Polythetic
Query Language	Artificial	Natural
Query Specification	Complete	Incomplete
Items wanted	Matching	Relevant
Error response	Sensitive	Insensitive

Tabelle 1: Van Rijsbergens Abgrenzung von Daten Retrieval und Information Retrieval

Wie in Tabelle 1 dargestellt, liefert das Daten Retrieval, zu dem relationale Datenbanken zählen, immer exakte Ergebnisse. Information Retrieval zeigt die am besten passenden Ergebnisse an. Diese Ergebnisse werden absteigend sortiert und gelistet. Es werden Dokumente angezeigt, die mit recht hoher Wahrscheinlichkeit interessant für den Anwender sein werden. Unter Inference versteht man die Fähigkeit, aus vorhandenem Wissen Schlussfolgerungen zu ziehen. Dazu zählt Deduction und Induction. Deduction bedeutet, aus Allgemeinem das Besondere herleiten. Der umgekehrte Fall findet sich bei der Induction, hier werden allgemeine Regeln aus Einzelfällen gezogen. In einem deterministic Model läuft jedes Ereignis nach feststehenden Gesetzen ab. Dadurch kann auf jede Anfrage deterministisch auf ein Ergebnis geschlossen werden. Bei einem probabilistischen Modell wird ein Ergebnis mit Wahrscheinlichkeitsrechnung bestimmt.

In relationalen Datenbanken wird die künstliche Sprache Structured Query Language, abgekürzt SQL, eingesetzt. Information Retrieval setzt auch zum Teil künstliche Sprachen wie die Booleschen Operatoren AND,OR und NOT ein. In der Regel werden die Anfragen in natürlicher Sprache formuliert. Die gewünschten Ergebnisse sind im Daten Retrieval immer exakt. Im Fall eines Information Retrieval-Systems sind gefundene Dokumente für einen Anwender mehr oder weniger relevant. Es werden ähnlich der Anfrage passenden Ergebnisse ausgegeben. Durch Ungenauigkeiten in der Anfragen und unvollständiges Wissen ist das Information Retrieval Fehlertoleranter. Leichte Fehler in den Ergebnissen werden toleriert, wenn dadurch die Laufzeit erheblich verbessert wird. Im Gegensatz dazu sind Daten

Retrieval-Systeme bei Fehlverhalten sehr sensibel und Fehler werden nicht toleriert. Im nächsten Abschnitt wird die Funktionsweise eines Information Retrieval-Systems beschrieben.

2.2 Funktionsweise eines Information Retrieval-Systems

Ein Information Retrieval-System kann als ein System beschrieben werden, das aus einer Menge von Dokumenten und einer Menge von Suchanfragen besteht. Zwischen den beiden Mengen gibt es eine Ähnlichkeitsfunktion, die die Aufgabe hat die beiden Mengen miteinander zu vergleichen. Eine Indexierungssprache liegt über der Ähnlichkeitsfunktion und dient als Übersetzer für die jeweiligen Prozesse. Man kann die Indexierungssprache als eine gemeinsame Sprache zwischen der Anfrage und den vorhandenen Dokumenten ansehen. Denn neue Dokumente müssen mit festgelegten Regeln, in dem Fall mit der Indexierungssprache, zu der Menge der bereits existierenden Dokumente hinzugefügt werden. Das gleiche geschieht mit einer Anfrage. Diese wird auch in die Indexierungssprache übersetzt. Die Beziehungen zwischen den einzelnen Komponenten sind in Abbildung 2 dargestellt.

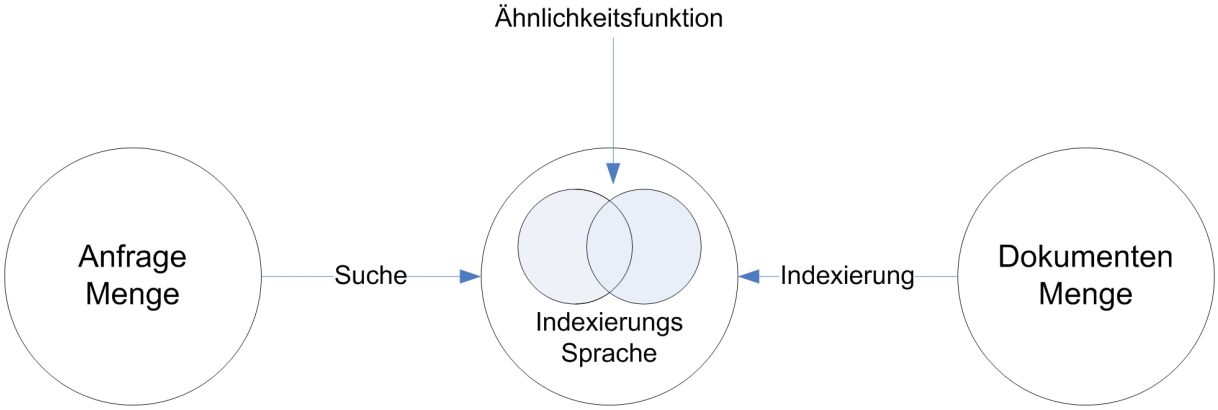

Abbildung 2: Funktionsweise von Information Retrieval nach Salton

Die Übersetzung der Dokumente in die Indexierungssprache kann manuell oder automatisch erfolgen. Welche Dokumente für die jeweiligen Suchanfragen relevant sind, wird also nicht direkt, sondern über die Repräsentation der Suchanfragen und Dokumente in der Indexierungssprache bestimmt. Die Begriffe der Indexierungssprache stehen entweder vorher fest (kontrolliert) oder sie stammen direkt aus dem Text der Dokumente und Suchanfragen

(unkontrolliert). Unabhängig von der Indexierungssprache wird angenommen, dass sich Dokumente durch eine Menge von Begriffen der Indexierungssprache repräsentieren lassen. Ein Begriffsauswahlverfahren entscheidet über die Identifikation eines Dokuments im Index und nicht der Originaltext.

2.3 Retrieval-Modelle

In diesem Abschnitt werden verschiedene Retrieval-Modelle vorgestellt. Ein Retrieval-Modell ist nach Baeza-Yates und Ribeiro-Neto [RIC99] (Seite 23) folgendermaßen definiert:

Definition: Ein Information Retrieval-Modell ist ein Quadrupel $[\boldsymbol{D}, \boldsymbol{Q}, F, R(q_i, d_j)]$

- **D** Menge logischer Sichten auf Dokumente (Repräsentationen)
- **Q** Menge logischer Sichten auf Informationswünsche eines Anwenders (Queries)
- F Modellierungsrahmen (Framework) für Dokumentrepräsentation **D**, Queries **Q** und der Beziehung zwischen **D** und **Q**
- $R(q_i, d_j)$ Rankingfunktion ordnet Query $q_i \in \boldsymbol{Q}$ und Dokument $d_j \in \boldsymbol{D}$ einen Wert zu. Definiert Reihenfolge der Dokumente bezüglich Query q_i .

Ein Modell besteht aus der Repräsentation der Dokumente und der Repräsentation von Informationswünschen eines Anwenders. Diese Repräsentationen werden mit Hilfe eines Frameworks erstellt, das wiederum eine Rankingfunktion unterstützen sollte. Beispielsweise besteht das Framework im Booleschen Modell aus einer Menge von Dokumenten und den booleschen Operatoren. Weitere Modelle wie Vektormodell und Probabilistisches Modell gehören zu den klassischen Modellen des Information Retrieval. Des Weiteren existieren Erweiterungen der klassischen Modelle. Dazu haben Baeza-Yates und Ribeiro-Neto eine Klassifikation der Modelle erstellt, die in der Abbildung 3 dargestellt ist.

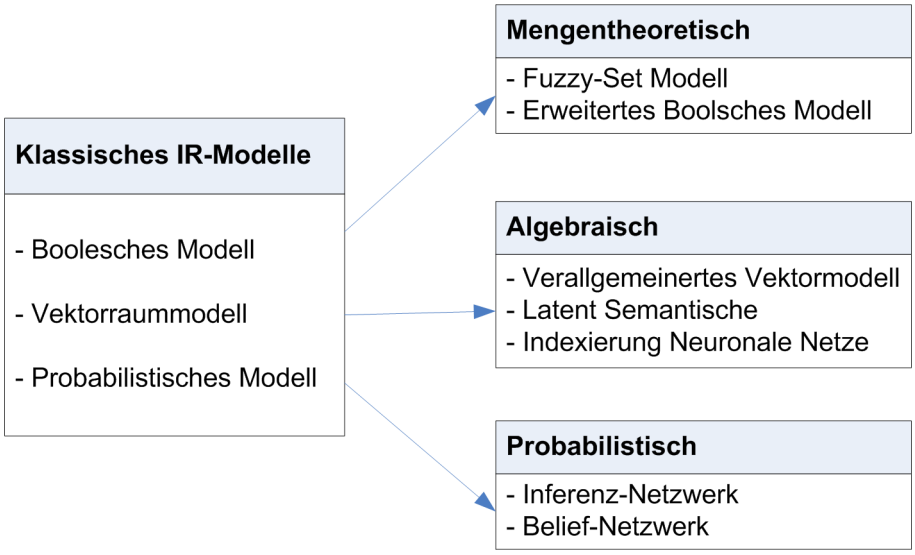

Abbildung 3: Übersicht der Information Retrieval-Modelle nach Baeza-Yates

Die Repräsentation aller Dokumente im IR-System umfasst nicht alle Wörter, die in den einzelnen Dokumenten vorkommen. Jedes Dokumente wird mit einer Menge von Schlüsselwörtern beschrieben, die Indexterme genannt werden. Ein Indexterm ist ein Wort bzw. ein Schlagwort, das im Dokument vorkommt. Mehrere dieser Terme können den Inhalt eines Dokuments eindeutig umschreiben. Allgemein bestehen Indexterme aus Nomen, weil diese den Inhalt eines Dokuments besser beschreiben. Es werden nicht alle Terme benutzt, da z.B. Terme, die mehrmals in allen Dokumenten vorkommen, nutzlos sind, da eine Suchanfrage damit nicht begrenzt werden kann. Kommt ein Term nur in wenigen Dokumenten vor, so kann dieser Dokumente besser voneinander abgrenzen. Dadurch wird deutlich, dass Indexterme eine unterschiedliche Gewichtung haben.

Nicht nur die Häufigkeit der Terme ist entscheidend für die Relevanz, der als Ergebnis angezeigten Dokumente, sondern auch das vom IR-System unterstütze Modell. So kann beispielsweise im Booleschen Modell keine Relevanz bestimmt werden. Der Begriff der Relevanz hat im Retrieval eine große Bedeutung. Relevanz ist eine Beurteilung von Retrieval-Ergebnissen. Dabei wird überprüft, in wie weit die angezeigten Dokumente der Anfrage entsprechen. Wolfgang G. Stock [WOL07] beschreibt Relevanz wie folgt und unterscheidet zwischen objektivem und subjektivem Informationsbedürfnis.

„Ein Dokument (bzw. das darin enthaltene Wissen) ist zur Befriedigung eines Informationsbedarfs relevant,

- *wenn es objektiv zur Vorbereitung einer Entscheidung dient oder*
- *wenn es objektiv eine Wissenslücke schließt oder*
- *wenn es objektiv eine Frühwarnfunktion erfüllt.*

Pertinenz bezieht den konkreten Nutzer in die Betrachtung ein. Zentral ist dabei das kognitive Modell des Nutzers. Insofern ist ein Dokument bzw. das Wissen für einen Nutzer zur Befriedigung eines Informationsbedürfnisses pertinent,

- *wenn es subjektiv zur Vorbereitung einer Entscheidung dient oder*
- *wenn es subjektiv eine Wissenslücke schließt oder*
- *wenn es subjektiv eine Frühwarnfunktion erfüllt.“*

Die Relevanz eines Dokuments ist Abhängig vom implementierten Retrieval-Modell im IR-System. Es folgt eine detaillierte Beschreibung des Booleschen Modells und des Vektorraummodells. Eine ausführliche Beschreibung des Probabilistischen Modells und der erweiterten Modelle wird u.a. von Baeza-Yates und Ribeiro-Neto behandelt. Deren detaillierte Beschreibung würde im Rahmen dieses Grundlagen-Kapitels zu weit führen.

2.3.1 Boolesches Modell

Das Boolesche Modell ist eine einfache Form des Information Retrieval. Es basiert auf der Mengenlehre und arbeitet im Wesentlichen mit den Operatoren Schnittmenge, Vereinigungsmenge und Komplementärmenge. Die einzelnen Operatoren werden in der Anfrage wie folgt formuliert: Für die Schnittmenge wird das AND ($\cap$), für die Vereinigungsmenge das OR ($\cup$) und für die Komplementärmenge das NOT ($\neg$) benutzt.
Die Abbildung 4 zeigt die Operationen im Booleschen Modell.

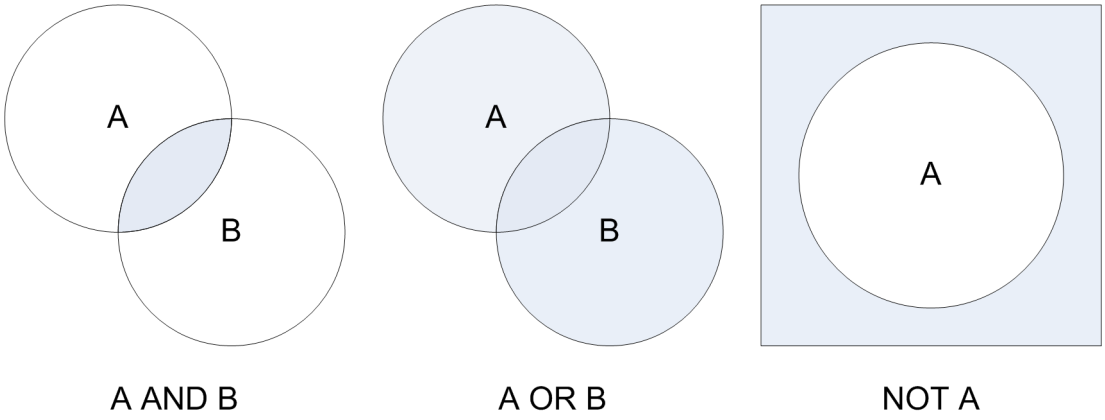

Abbildung 4: Boolesche Operationen

Reginald Ferber [REG03] definiert das Boolesche Retrieval wie folgt:

Definition Boolesches Retrieval

Sei $\boldsymbol{D}$ *eine Menge von Dokumenten und* $\boldsymbol{t : D \rightarrow T,\ t(d) = t_i}$ *ein Attribut.*

Die Menge

$$\boldsymbol{D_{t,t_i} = t^{-1}(\{t_i\}) = \{d \in D \mid t(d) = t_i\}}$$

der Dokumente, bei denen das Attribut $\boldsymbol{t}$ *den Wert* $\boldsymbol{t_i}$ *annimmt, die also durch den Attributwert* $\boldsymbol{t_i}$ *charakterisiert sind, wird als Ergebnismenge der elementaren Anfrage* $\boldsymbol{(t, t_1)}$ *bezeichnet. Sind in einer Anfrage zwei Attribut-Wert-Paare durch einen booleschen Operator verknüpft, wird damit die entsprechende Verknüpfung der Dokumentenmengen bezeichnet:* $\boldsymbol{(t, t_1)\ \mathbf{AND}\ (s, s_1)}$ *bezeichnet den Durchschnitt* $\boldsymbol{D_{t,t_1} \wedge D_{s,s_1}}$, *der Ausdruck* $\boldsymbol{(t, t_1)\ \mathbf{OR}\ (s, s_1)}$ *die Vereinigung* $\boldsymbol{D_{t,t_1} \vee D_{s,s_1}}$ *und der unäre Operator* $\boldsymbol{NOT(t, t_1)}$ *das Komplement* $\boldsymbol{D \setminus D_{t,t_1}}$.

Eine Anfrage kann demnach folgendermaßen formuliert werden: „Datenbanken AND xml“. Mit dieser Anfrage möchte der Anwender Dokumente finden, welche die Begriffe Datenbank und xml beinhalten. Das Ergebnis der Anfrage ist eine Teilmenge, die mit booleschen Operatoren weiter bearbeitet werden kann. Damit lassen sich beliebig viele geschachtelte Ausdrücke erstellen, die eine exakte Ergebnismenge liefern.

2.3.2 Vektorraummodell

Das Vektorraummodell ist ein Ranking-Modell, in dem Dokumente und Anfragen als Punkte in einem mehrdimensionalen Koordinatensystem abgebildet werden. Reginald Ferber [REG03] definiert ein Vektorraummodell wie folgt:

Definition Vektorraummodell

Sei $T=\{t_1,\dots,t_n\}$ *eine endliche Menge von Termen und* $D=\{d_1,\dots,d_m\}$ *eine Menge von Dokumenten. Für jedes Dokument* $d_i \in D$ *sei zu jedem Term* $t_k \in T$ *ein Gewicht* $w_{i,k} \in R$ *gegeben. Die Gewichte des Dokuments* d_i *lassen sich zu einem Vektor* $w_i=(w_{i,1},\dots,w_{i,n}) \in R^n$ *zusammenfassen. Dieser Vektor beschreibt das Dokument im Vektorraummodell: Er ist seine Repräsentation und wird Dokumentvektor genannt.*

Auch Anfragen (Queries) werden durch Vektoren $q \in R^n$ *repräsentiert. Wie bei der Repräsentation der Dokumente wird die Anfrage durch eine Menge gewichteter Terme dargestellt. Der Vektor der Gewichte wird Anfragevektor oder Query-Vektor genannt.*

Schließlich sei eine Ähnlichkeitsfunktion $s: R^n \times R^n \rightarrow R$ *definiert, mit der jedem Paar aus zwei Vektoren* $x, y \in R^n$ *ein reeller Ähnlichkeitswert* $s(x,y)$ *zugewiesen wird.*

Im Gegensatz zum Booleschen Modell hat das Vektorraummodell eine Ähnlichkeitsfunktion, die relevante Dokumente der Suchanfrage präsentiert. Die Ähnlichkeit zwischen der Anfrage und einem Dokument ergibt sich aus der räumlichen Nähe. Diese Nähe wird mit einer Winkelfunktion berechnet. Salton und McGill [SAL83] (Seite 129) beschreiben diese Funktion folgendermaßen:

„Die Ähnlichkeit zwischen zwei Dokumentvektoren lässt sich durch eine Funktion bestimmen, die umgekehrt proportional zum Winkel zwischen den beiden Vektoren ist. Sind zwei Dokumentvektoren genau gleich, werden die entsprechenden Vektoren übereinander liegen, so dass der Winkel zwischen den Vektoren 0 ist.“

Zur Berechnung der Ähnlichkeit nach dem jeweiligen Winkel verwendet Salton den Cosinus:

$$\cos(Dok_i, Anfrage_j) = \frac{\sum_{k=1}^{t} (TERM_{ik} \cdot ATERM_{jk})}{\sqrt{\sum_{k=1}^{t} (TERM_{ik})^2 \cdot \sum_{k=1}^{t} (ATERM_{jk})^2}}$$

Demnach wird der Cosinus des Winkels zwischen Dokument und Suchanfrage in einem Vektorraum mit t Dimensionen berechnet. In Abbildung 5 ist ein dreidimensionaler Dokumentenraum dargestellt.

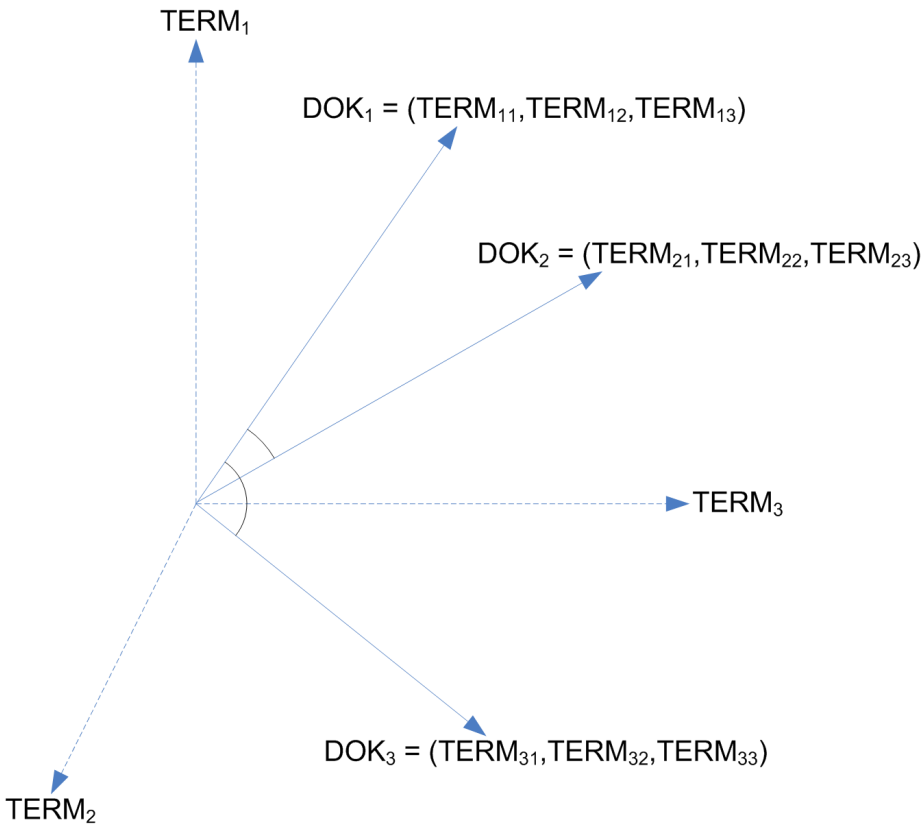

Abbildung 5: Vektorrepräsentation eines Dokumentraumes nach Salton

Die Dokumente werden durch drei Terme beschrieben. Jeder Term wird einer Dimension zugeordnet. Um die Position eines Dokuments im Vektorraum zu bestimmen, kann z.B die Häufigkeit der Terme, die in dem Dokument vorkommen, berechnet werden. Dabei werden

für die Relevanzbeurteilung der Dokumente reelle Zahlen zur Gewichtung verwendet.

Der einfachste Fall zur Berechnung der Häufigkeiten kann laut Ferber Reginald [REG03] wie folgt berechnet werden:

$$w_{i,j} = h(i,j)$$

„Dabei bezeichnet ***h(i,j)*** *die Häufigkeit von Term* t_j *im Dokument* d_i *. Andere Formeln beschränken die Gewichte auf ein Intervall und dämpfen den Einfluss sehr häufiger Terme, wie z.B. die Formel „*

$$w_{i,j} = \frac{h(i,j)}{1+h(i,j)}$$

Eine Suchanfrage wird in Vektoren umgewandelt und kann in natürlicher Sprache formuliert werden. Die Abfragevektoren und Dokumentenvektoren werden paarweise miteinander verglichen. Die dabei resultierenden Ähnlichkeitswerte können für die Erstellung einer Rangfolge genutzt werden.

2.4 Retrieval-Prozess

Der Retrieval-Prozess ist ein Ablauf, der notwendig ist, um Dokumente durchsuchen zu können. Dieser Ablauf teilt sich in vier Schritte auf: Textanalyse der Dokumente, das Indexieren an sich, die Suche im Index und anschließend die Ergebnispräsentation. Eine Suche in Dokumenten kann auf unterschiedliche Weise erfolgen. Dabei unterscheidet man zwischen einer direkten Suchen in Textdokumenten oder einer Suche mit Hilfe eines erstellten Index.

Die direkte Suche, auch „Online Suche“ genannt kann nur bei Textdokumenten effizient eingesetzt werden, wenn diese klein sind oder sich häufig verändern. Die Suche mit Hilfe eines Index wird bei großen Datenmengen eingesetzt und aufgrund ihrer Struktur auch schnell. Die häufigste Form eines Index sind invertierte Dateien, „inverted files“ oder „inverted indices“ genannt. Abbildung 6 zeigt den allgemeinen Retrieval-Prozess mit Textanalyse nach Baeza-Yates und Ribeiro-Neto.

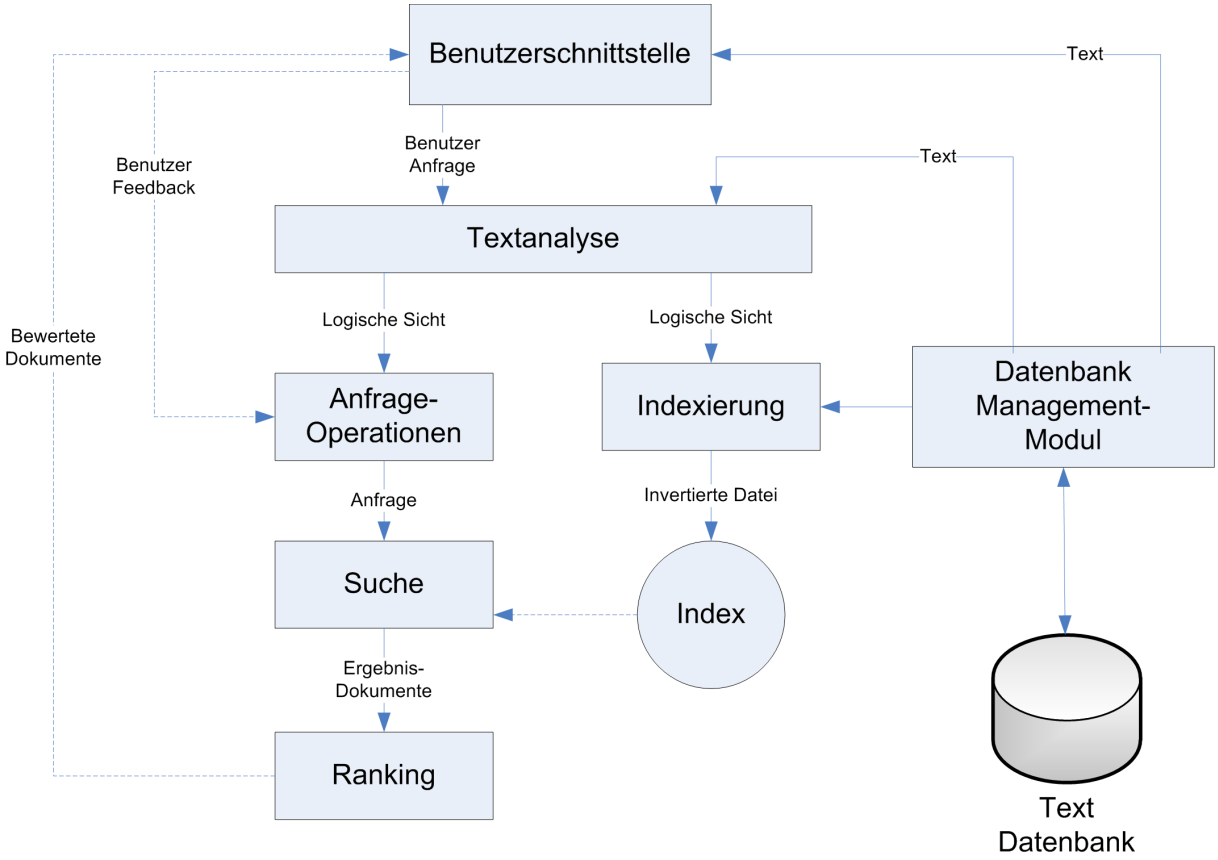

Abbildung 6: Retrieval-Prozess nach Baeza-Yates und Ribeiro-Neto

2.4.1 Datenaufbereitung und Textanalyse

Bei diesem Schritt werden Textdokumente in Terme zerlegt. Die Terme durchlaufen anschließend mehrere Filter, um den Datenumfang zu reduzieren. Wie in Abbildung 7 zu sehen ist, sind die Filter hintereinander angeordnet, damit jeder Term genau überprüft werden kann, ob dieser für die Informationsgewinnung des Textdokuments geeignet ist. Anschließend wird der Term automatisch oder manuell, also nach einer zusätzlichen Überprüfung, in den Index hinzugefügt.

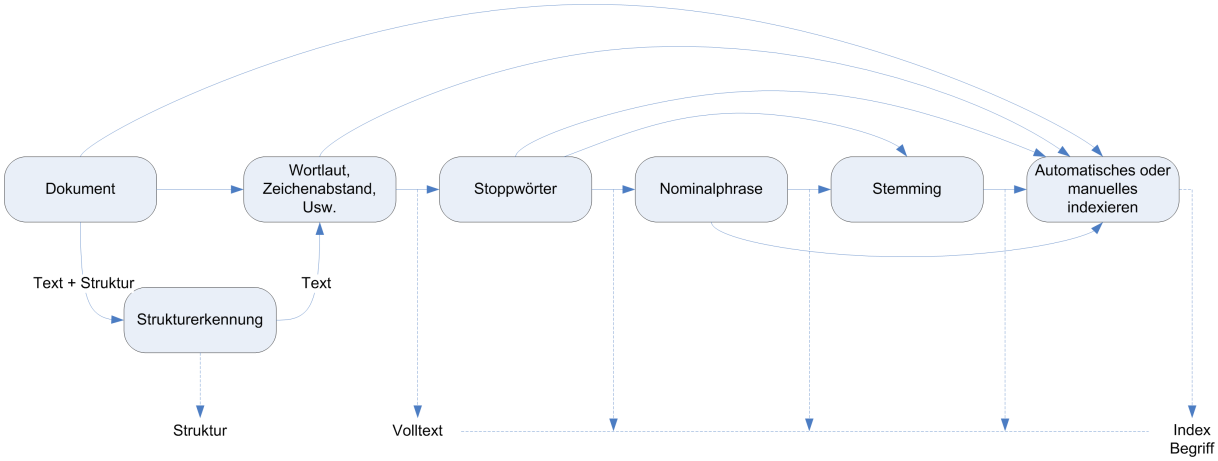

Abbildung 7: Retrieval-Prozess nach Baeza-Yates und Ribeiro-Neto

Analyse und Term-Erstellung

Als Erstes wird das Textdokument in Terme zerlegt. Dabei werden die Terme so vorbereitet, dass sie in weiteren Filtern bearbeitet werden können. So werden Bindestriche und Satzzeichen eliminiert. Des Weiteren werden die Terme in Klein- oder Großschreibung abgeändert. Ziffern werden gesondert behandelt und als Terme weitergereicht, da diese wichtige z.B. historische Daten beinhalten und zur Bedeutung des Inhaltes beitragen.

Stoppwörter

Stoppwörter sind Terme, die bei der Indexierung nicht aufgenommen werden. Sie kommen häufig vor und sind für die Interpretation eines Dokuments nicht wichtig. Stoppwörter-Eliminierung ist sprachspezifisch. Typische deutsche Stoppwörter sind die bestimmten Artikel „der, die, das“, die unbestimmten Artikel „ein, einer, eine“, Konjunktionen wie „und, oder, auch“, Präpositionen wie „an, in, von“ und die Negation „nein“. Ginge man nur nach den Regeln der Stoppwörter-Eliminierung, dann würde man das Zitat von William Shakespeare „be or not to be“ nicht mehr wiederfinden, da dieser Beispielsatz aus Englischen Stoppwörtern besteht.

Nominalphrase

Betrachtet man die Sprache der Textdokumente, so kann man bei den Nomen erkennen, dass diese mehr die Semantik eines Textes wiedergeben. Deshalb werden Wortarten wie Adjektive,

Adverbien, Verben, Pronomen und andere eliminiert. Kommen Nomen mehrfach hintereinander vor, beispielsweise „Computer Wissenschaft“ so werden diese als Nominalphrase indexiert.

Stemming

Grundformreduktion im englischen Stemming genannt, ist eine Methode, um Wörter in deren lexikalischen Kern zurückzuführen. Dabei werden Wörter wie Stricknadel, Strick, verstrickt und bestricken auf das Stammwort „strick“ reduziert. Für Stemming-Verfahren werden Algorithmen eingesetzt, die es für verschiedene Sprachen gibt. Einer der bekanntesten Algorithmen ist der Porter-Stemmer-Algorithmus. Allerdings arbeitet er nicht genau, da manche Wortstämme nicht dem linguistischen Wortstamm entsprechen. Dabei können Wörter zu viel (Overstemming) oder zu wenig (Understemming) abgeschnitten werden. Das lässt sich aber bei allen Stemming-Algorithmen nicht vermeiden.

Auswahl der Indexierung

Es gibt drei unterschiedliche Arten von Indexierungs-Verfahren. Das manuelle, das automatische und das semiautomatische Verfahren. Das klassische Indexierungs-Verfahren ist das manuelle Verfahren. Die manuelle Indexierug hat die Aufgabe, Dokumente inhaltlich zu analysieren und zu erschließen. Dabei werden folgende Arbeitsschritte gemacht: bei der Inhaltsanalyse wird der Inhalt eines Dokuments vom Menschen erfasst, um anschließend die Repräsentation dieser Inhalte durch die sprachlichen Elemente einer Indexierungssprache zu erstellen. Dabei wird eine Zusammenfassung geschrieben, die den Inhalt des Dokuments wiedergibt. Dieses Verfahren wird bei Dokumenten eingesetzt, die nicht in elektronischer Form vorliegen.

Die automatische Indexierung ist ein Verfahren, bei dem Dokumente automatisch analysiert werden. Terme werden aus den Dokumenten extrahiert. Das Verfahren läuft vollautomatisch. Dabei wird die Beschreibung des Inhalts vom Computer ermittelt und erfasst. Die Inhaltsanalyse basiert auf statistischen bzw. linguistischen Kriterien sowie auf Mustererkennung. Die Bedeutung des Inhalts kann zur Zeit nur durch den Menschen erfasst werden.

Die semiautomatische Indexierung erfolgt wie die automatische Indexierung, wobei die Beschreibung des Inhaltes nicht automatisch erfasst, sondern nur vorgeschlagen wird. Die Vorschläge werden anschließend von einem Menschen kontrolliert und gegebenenfalls ausgebessert und erfasst.

2.4.2 Datenstrukturen von Information Retrieval-Systemen

In dem Prozess der Textanalyse werden Terme aus Dokumenten herausgefiltert. Diese werden im Folgenden als Indexterme genannt. Diese müssen in eine geeignete Datenstruktur abgespeichert werden, die Index genannt wird. Dabei werden nicht nur die Indexterme, sondern auch deren Position im Dokument und die Identifikation des Dokuments mit abgespeichert. Die Indexterme werden nur einmal pro Dokument, aber mit deren unterschiedlichen Positionen aufgenommen. Man versucht die Struktur so klein wie möglich zu halten und die Suche zu beschleunigen.

Ricardo Baeza-Yates [RIC99] beschreibt :

> „build date struktur over the text (called indicies) to speed up the search“

Vier verwendete Verfahren der Indexerstellung werden in diesem Kapitel beschrieben. Dazu zählen Invertierte Datei, n-Gramme, Signatur-Dateien und Suffix-Bäume/-Felder. Das am meisten verwendete Verfahren ist die invertierte Datei.

Invertierte Datei

Die Struktur einer Invertierten Datei ist wie ein Index aufgebaut, den man aus Büchern kennt. Für bestimmte Begriffe wird im Index eine Seitennummer angegeben, mit der man schnell die jeweilige Stelle finden kann. Überträgt man das Prinzip auf Dokumente, so kann auch ein Index für alle vorhandenen Dokumente erstellt werden. Dabei werden Begriffe aus den jeweiligen Dokumenten genommen und mit den entsprechenden Verweis auf das Dokument sowie der Seitenanzahl im Index abgespeichert. Abbildung 8 veranschaulicht das Prinzip.

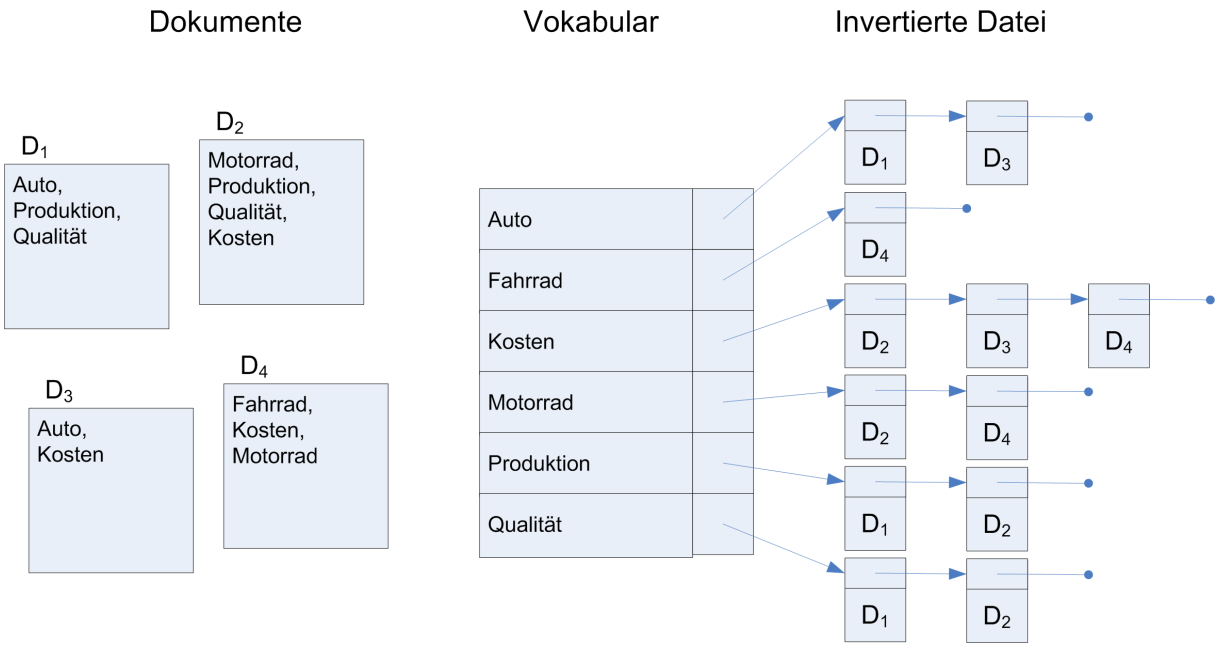

Abbildung 8: Invertierte Datei nach Henrich

Auf der linken Seite sind Dokumente dargestellt, welche Wörter enthalten, die durchsuchbar gemacht werden sollen. Auf der rechten Seite werden diese Dokumente in einer invertierten Datei organisiert. Die in der Mitte dargestellte Tabelle, die Begriffe beinhalten, welche in allen links dargestellten Dokumenten vorkommen. Die Tabelle enthält keine Duplikate. Jede Tabellenzeile besteht aus einem Begriff und einem Verweis auf eine invertierte Liste. Jedem Begriff wird eine invertierte Liste zugeordnet. In dieser werden Verweise auf die Dokumente gespeichert. Sucht man nach einem Wort, so wird das Information Retrieval-System erst in der Vokabular-Tabelle nachschauen, um an die Liste der Dokumentverweise zu gelangen. Die Liste der Verweise wird als Ergebnis präsentiert.

Der Begriff „invertierte Datei“ wird für die Vokabular-Tabelle und die einzelnen invertierten Listen verwendet. Man spricht von invertierten Dateien, da die invertierten Listen als Dateien abgespeichert werden.

n-Gramme

Im vorherigen Verfahren der invertierten Dateien wurden die Textdokumente mit linguistischen Mitteln bearbeitet. Das bedeutet, dass man auf die jeweilige Sprache der Textdokumente eingegangen ist und diese bei der Textanalyse beachtet hat. Dabei wurden die

Worte auf Stoppworte und Phrasenbildung untersucht und bearbeitet, sowie auf deren Grundform reduziert. Erst dann kann man die Tabelle mit Vokabular aufbauen. Möchte man aber sprachunabhängig die Worte bearbeiten, so arbeitet man mit n-Grammen. Man betrachtet keine Worte mehr als solche, sondern bringt die Zeichenfolge eines Textdokuments auf die Länge n. n steht für eine Zahl zwischen 1 und in der Regel 6.

Geht man z.B von einer Variante 2-Gramme, auch Bigramme genannt, aus, so erhält man bei der Zerlegung des Textes „INFORMATION RETRIEVAL“ folgende Bigramme vor.

„*I, IN, NF, FO, OR, RM, MA, AT, TI, IO, ON, N*“,

„*R, RE, ET, TR, RI, IE, EV, VA, AL, L*“

(Stern steht für ein Leerzeichen)

Bei Retrievalsystemen die n-Gramme einsetzen hat man den Vorteil, dass man nicht nur sprachunabhängig ist, sondern auch die maximale Anzahl der Indexterme bestimmen kann. Wenn man von 26 Buchstaben + Leerzeichen ausgeht, so erhält man mit 27^n die maximale Größe der Indexeinträge. Bei Bigramme beträgt die maximale Größe also $27^2 = 729$ Indexterme.

Das n-Gramme Verfahren wird auch bei Korrekturprogrammen für Tippfehler eingesetzt.

Signaturen

In den oben beschriebenen Verfahren der invertierten Dateien und n-Gramme wurden Worte entweder durch linguistische Mittel oder durch Zerlegung der Worte in Teilworte umgewandelt. Bei dem Verfahren der Signaturen werden Worte aus Dokumenten in Bitstrings einer festen Länge abgebildet. Henrich [HEN08] definiert Signatur folgendermaßen :

Definition Signatur

Sei F eine natürliche Zahl. Eine Signatur der Länge F ist ein Bitstring $s = b_1 b_2 .. b_F$ mit $b_i \in \{0;1\}$ für $1 \leq i \leq F$.

Signaturen werden mit Hilfe von Hash-Funktionen erzeugt. Hash-Werte haben einen wesentlichen Vorteil gegenüber Zeichenketten, das sie schneller verglichen werden können.

Folgendes Beispiel soll eine einfache Implementierung einer solchen Hash-Funktion darstellen.

```
hash := 0;
for i := 1 to l do
     hash := (hash + wortzeichen_pos_i) * 157;
end
hash := hash mod 2^F;
```

Im Beispiel werden alle Zeichen eines l langen Wortes durch Addition, Multiplikation mit der Primzahl 157 und anschließend einer Modulo Operation in eine Signatur umgewandelt. Die Hash-Werte haben aber einen Nachteil: bei der Berechnung entstehen künstliche Homonyme. Ein Homonym ist ein Wort, das für verschiedene Begriffe steht. Das bedeutet, dass für verschiedene Wörter dieselbe Signatur berechnet wird. Somit können bei einer Suche Ergebnisse angezeigt werden, die keine Bedeutung für die Abfrage des Benutzers haben. Die fehlerhaften Dokumente, die durch den Effekt der künstlichen Homonyme angezeigt werden, nennt man „false drops“. Die Suchanfrage wird als Hash-Wert berechnet und mit der Menge aller Hash-Werte verglichen. Der Vergleich basiert auf Zahlen und nicht auf Zeichenketten, dadurch erhöht sich die Geschwindigkeit bei der Suche nach relevanten Dokumenten.

Suffix Array und Suffix Baum

Bei den Begriffen „Suffix Baum“ und „Suffix Array“ handelt es sich um Verfahren zur Erstellung einer Datenstruktur, die aus einer Zeichenkette, beispielsweise eines Textdokuments, Suffixe erstellt. Ein Suffix ist eine Nachsilbe, die einem Wortstamm angehängt ist. Im Kontext des Suffix Array werden Suffixe aus der Menge aller Zeichenketten eines Dokuments erstellt. Dabei betrachtet man das Dokument selbst als eine lange Zeichenkette. Bei der Zeichenkette wird sukzessiv das erste Zeichen entfernt. Dabei entstehen die sogenannten Suffixes. Folgendes Beispiel verdeutlicht das Prinzip.

Betrachtet man die Zeichenkette „Raumfahrt“, so besteht die Menge aller Suffixe aus den Zeichenketten „Raumfahrt“, „aumfahrt“, „umfahrt“, „mfahrt“, „fahrt“, „ahrt“, „hrt“, „rt“ und

„t“. Anschließend werden die Suffixe nach lexikographischer Reihenfolge sortiert: „ahrt“, „aumfahrt“, „fahrt“, „hrt“, „mfahrt“, „raumfahrt“, „rt“, „t“ und „umfahrt“. Ein Suffixarray ist ein Array mit Verweisen auf die Positionen der Suffixe im Originaltext. Für die Zeichenkette „Raumfahrt“ besteht der Suffixarray aus folgenden Positionen {6,2,5,7,4,1,8,9,3}, weil „ahrt“ beim sechsten Zeichen beginnt, „aumfahrt“ beim zweiten Zeichen, „fahrt“ beim fünften Zeichen, „hrt“ beim siebten Zeichen, „mfahrt“ beim vierten Zeichen, „raumfahrt“ beim ersten Zeichen, „rt“ beim achten Zeichen, „t“ beim neunten Zeichen und „umfahrt“ beim dritten Zeichen beginnt.

Bei längeren Texten werden die Suffixe nicht aus jedem Zeichen erstellt. Ein Wortanfang wird bei jedem sukzessiven Anlauf als Startposition festgelegt. Dadurch nähert man sich an das Indexierungs-Verfahren von invertierten Dateien an. Der Unterschied zwischen Suffix Array und Suffix Baum ist der, dass das Array eine kompaktere Speicherform hat als der Baum. Bei dem Suffix Array werden lediglich, wie auf der vorherigen Seite zu sehen ist, nur die Positionen der Suffixe abgespeichert. Wie in der Abbildung 9 zu sehen ist, besteht ein Suffix Baum aus Kanten und Blättern. Dabei stehen Kanten für Zeichen und Blätter für Verweise.

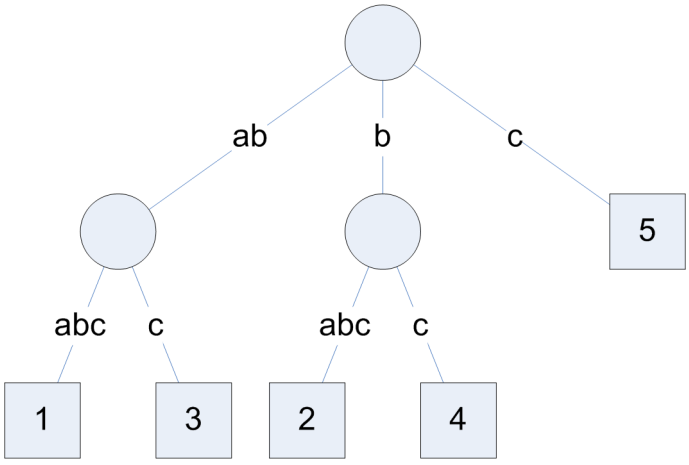

Abbildung 9: Suffix Baum für "ababc"

Alle Pfade von der Wurzel zu den Blättern ergeben alle Suffixe. Betrachtet man nur die Blätter des Baumes in Abbildung 9, so erkennt man, dass diese Werte dem Suffix Array für „ababc“ entsprechen. In diesem Fall (1,3,2,4,5).

2.4.3 Suchanfrage

Eine Suche nach Dokumenten kann in Suchanfrage, den eigentlichen Suchprozessor eines Information Retrieval-Systems und die Ergebnisse unterteilt werden. Die ersten beiden Punkte werden in diesem Abschnitt näher erläutert. Der letzte Punkt wird gesondert im nächsten Abschnitt beschrieben.

Eine Suche beginnt mit der Formulierung einer Suchanfrage, die vom Benutzer oder von einem Programm erstellt wird. Abhängig vom unterstützten Modell im IR-System können verschiedene Suchanfragen formuliert werden. Es können nicht nur einzelne Suchbegriffe- sondern auch Wortkombinationen oder der Ausschluss von Begriffen als Anfrage gestellt werden. Die unterschiedlichen Varianten werden im Verlauf des Kapitels detailliert beschrieben. Eine formulierte Suchanfrage wird über den Suchprozessor ausgeführt. Dabei durchläuft eine konkrete Suchanfrage die gleichen Filter, die beim Indexieren von Dokumenten angewendet worden sind. Es ist notwendig, die gleichen Filter anzuwenden, um die Ergebnisse bei einer Suche nicht zu verfälschen. Ein Suchprozessor hat die Aufgabe, die zu der Suchanfrage passenden Ergebnisse im Index zu finden und diese anschließend sortiert auszugeben. Somit ist ein Suchprozessor die eigentliche Suchkomponente und bildet eine Schnittstelle zwischen Anwender und dem Index eines Information Retrieval-Systems. Abbildung 10 zeigt die Beziehungen zwischen den Komponenten.

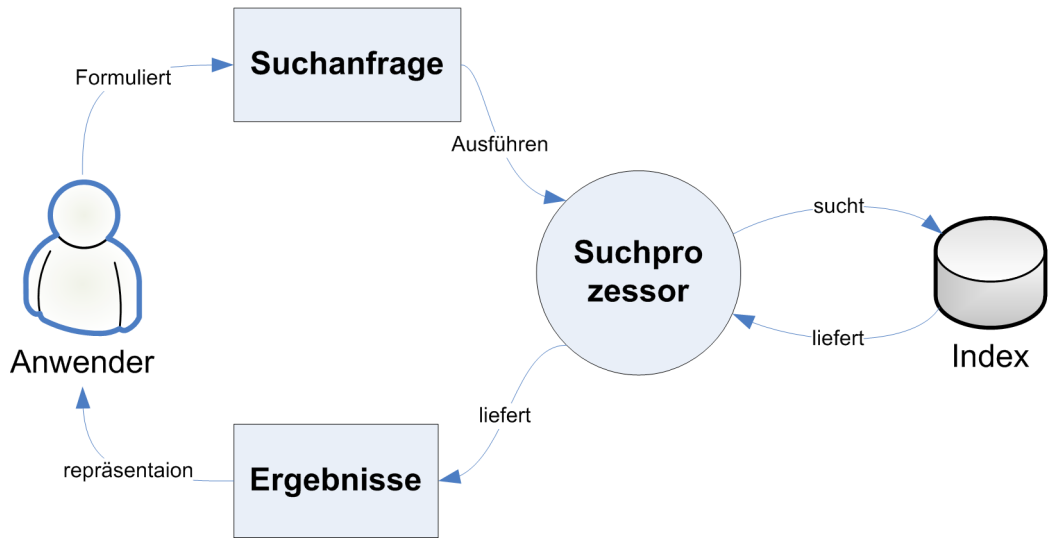

Abbildung 10: Beziehungen zwischen den Such Komponenten

Im Folgenden werden die verschiedenen Suchanfragen vorgestellt:

Single Word Anfrage

Die einfachste Suchanfrage kann mit einem Wort formuliert werden. Ein Wort besteht aus Buchstaben. Eine Single Word Anfrage kann nicht nur aus Buchstaben bestehen, sondern auch aus Zeichen, wie einem Bindestrich. z.B. „on-line“. Letztendlich wird in einem Retrieval-System festgelegt, wie eine Single Word Anfrage formuliert werden darf.

Phrasensuche

Bei der Phrasensuche wird die Suchanfrage in Anführungszeichen gesetzt, z.B. „Information Retrieval“. Dies wird sehr häufig bei der Suche nach Personen eingesetzt, den diese haben einen Vor- und zugehörigen Nachnamen. Erst durch das Zusammenfassen mit Anführungszeichen werden beispielsweise Personen wie „Albert Einstein“ erst eindeutig. Die Wortkombination wird in einem Retieval System als zusammengesetztes Wort gesucht.

Proximity Anfrage

Proximity ist der Abstand zwischen zwei Wörtern in einem Text. Dabei kann beispielsweise in einer Anfrage bestimmt werden, wie weit zwei Suchbegriffe voneinander entfernt sein dürfen.

Boolesche Anfrage

Bei der booleschen Suche muss der Benutzer seine Anfrage in eine Boolesche Anfrage-repräsentation formulieren. Diese besteht aus Booleschen Operatoren wie AND, OR und NOT. Dabei werden Begriffe mit den Operatoren miteinander verknüpft. Beispiel „Datenbanken AND XML“. Das Ergebnis der Beispielanfrage, gibt eine Menge der Dokumente aus, die die beiden Begriffe enthalten.

Pattern Matching

Pattern Matching basiert auf der Mustersuche. Es ist ein Verfahren, mit dem Strukturen, beispielsweise einen Teil eines Wortes mit einem vorgegebenen Muster identifiziert wird. Es

gibt unter anderem folgende Mustertypen: Prefixes, Suffixes, Substrings, Ranges und Regular Expressions. Beim Prefix wird der Anfang jeden Wortes daraufhin überprüft, ob das angegebene Muster passt. Ist z.B. der Prefix „info“ gegeben, so werden Wörter wie „information“, „informatik“ usw. gefunden. Beim Suffix wird das Ende jeden Wortes daraufhin überprüft, ob das angegebene Muster zutifft. Beim Substring wird keine Position festgelegt, an der ein Muster im Wort zutreffen soll, sondern es soll generell im Wort vorkommen. Beim Range geht es um einen Bereich von einem bis zum anderen Muster, also zwischen zwei Mustern. Der Bereich ist alphabetisch sortiert. So werden Wörter, die zwischen „Karo“ und „Peter“ liegen, gefunden. Bei den Regular Expressions handelt es sich um eine weitaus komplexere Mustersuche. Mit Hilfe von syntaktischen Regeln werden Zeichenketten gebildet. Dabei können die exakte Menge und die exakten Zeichen als Muster definiert werden z.B. [0-9]{4,5}. Dabei werden vier oder fünf Ziffern in Folge gefunden. Mögliche Ergebnisse „5678“, „54321“, jedoch nicht „abc“ oder „123“.

Suche in Strukturen

Die indexierten Dokumente wurden in Teilbereiche wie beispielsweise Titel, Autor, Zusammenfassung oder Erstellungsdatum aufgeteilt. Kennt der Anwender diese Struktur, so kann er gezielt die Suche auf bestimmte Bereiche eingrenzen. Z.B. können Dokumente gefunden werden, die der Autor „autor:Baeza-Yates“ geschrieben hat.

2.4.4 Suchergebnisse

Das Ergebnis einer Suchanfrage wird in den meisten Fällen als eine sortierte Liste präsentiert. Die Beurteilung der Ergebnisse kann in Präsentation und Informationsangebot unterteilt werden. Die Form der Präsentation von Suchergebnissen ist bei der Beurteilung eines Information Retrieval-Systems wichtig, denn eine einfache Form begünstigt den Zugriff auf relevante Informationen, die der Anwender benötigt. Das führt zur positiven Beurteilung des IR-Systems. Das Informationsangebot besteht aus allen Dokumenten, die einer Suchanfrage ähnlich sind. Dabei wird für jedes Dokument ein Relevanzwert berechnet und alle gefundenen Dokumente in absteigender Reihenfolge sortiert. Dokumente mit höchstem Relevanzwert stehen an erster Stelle der Liste und Dokumente mit geringstem Wert am Ende der Liste. Die meisten Ergebnislisten bestehen aus Verweisen, zu den tatsächlichen Dokumenten und einem Ausschnitt aus dem Originaltext, in dem die Suchbegriffe hervorgehoben werden. Zusätzlich

zu einer Ergebnisliste wird die Anzahl aller gefundener Dokumente zu einer Suchanfrage angezeigt. Diese Anzahl ist für die Berechnung von Kennzahlen wichtig. Ziel des Information Retrieval ist es, ausschließlich alle relevanten Dokumente des Informationsbedarfs eines Nutzers zu finden. Die Qualität einer Ergebnismenge wird durch die beiden Kennzahlen „Recall“ und Precision“ gemessen. [GER87] (Seite 174)

„Recall ist der Anteil relevanter Dokumente, die nachgewiesen wurden, während Precision der Anteil der nachgewiesenen Dokumente ist, die relevant sind.“

$$\textbf{\textit{Recall}} = \frac{\textbf{\textit{Zahl der nachgewiesenen relevanten Dokumente}}}{\textbf{\textit{Zahl aller relevanten Dokumente der Datenbank}}}$$

$$\textbf{\textit{Precision}} = \frac{\textbf{\textit{Zahl der nachgewiesenen relevanten Dokumente}}}{\textbf{\textit{Zahl aller nachgewiesenen Dokumente}}}$$

Beide Kennzahlen können Werte zwischen 0 und 1 annehmen. Ziel ist es, einen hohen Recall und eine hohe Precision zu erreichen.

Beispiel: Eine Suchanfrage liefert 200 Dokumente aus einem Gesamtbestand von 10.000.000 Dokumenten. 100 Dokumente sind im Gesamtbestand relevant. Im Ergebnis sind 40 davon gelistet.

Der Wert des Recall liegt bei 0,4 (40%) und der des Precision liegt bei 0,2 (20%).

3 Verwendete Technologien

Im Rahmen dieses Kapitels werden Technologien vorgestellt, die in dieser Arbeit eingesetzt werden. Die Apache Lucene-Bibliothek und die Eclipse Rich-Client-Platform werden im Folgenden vorgestellt. Die Apache Lucene-Bibliothek dient zum Erstellen von Volltextsuchmaschinen und die Eclipse Rich-Client-Platform wird zum Erstellen einer Benutzerschnittstelle eingesetzt.

3.1 Apache Lucene

Apache Lucene ist eine frei verfügbare Java-basierte Open Source Bibliothek, die unter der Apache Software Foundation entwickelt wird. Mit Hilfe dieser Bibliothek kann eine Volltextsuchmaschine entwickelt werden. Auf der Internetseite von Apache Lucene [LUC09], wird die Bibliothek wie folgt beschrieben:

„Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.“

Neben der Hauptentwicklung der Bibliothek in Java gibt es auch Portierungen in andere Programmiersprachen wie Perl, Python, C++, .NET und Ruby. Lucene wurde im März 2000 zum ersten Mal als Open Source Projekt veröffentlicht. Anschließend wurde das Projekt im September 2001 unter Apache Software Foundation Jakarta aufgenommen und ist somit unter der Apache Software Licence frei verfügbar. Im Folgenden Abschnitt werden typische Anwendungsgebiete von Lucene beschrieben.

3.1.1 Anwendungsgebiete

Ein typisches Anwendungsgebiet von Lucene ist das Indexieren von Dokumenten innerhalb einer Internetseite, um diese anschließend mit Hilfe einer Suchfunktion durchsuchen zu können. Weitere Anwendungsgebiete sind denkbar, beispielsweise die Integration einer Suchfunktion auf einer CD-ROM oder im Mail-Archiv. Abbildung 11 zeigt eine typische Anwendungsintegration mit Lucene.

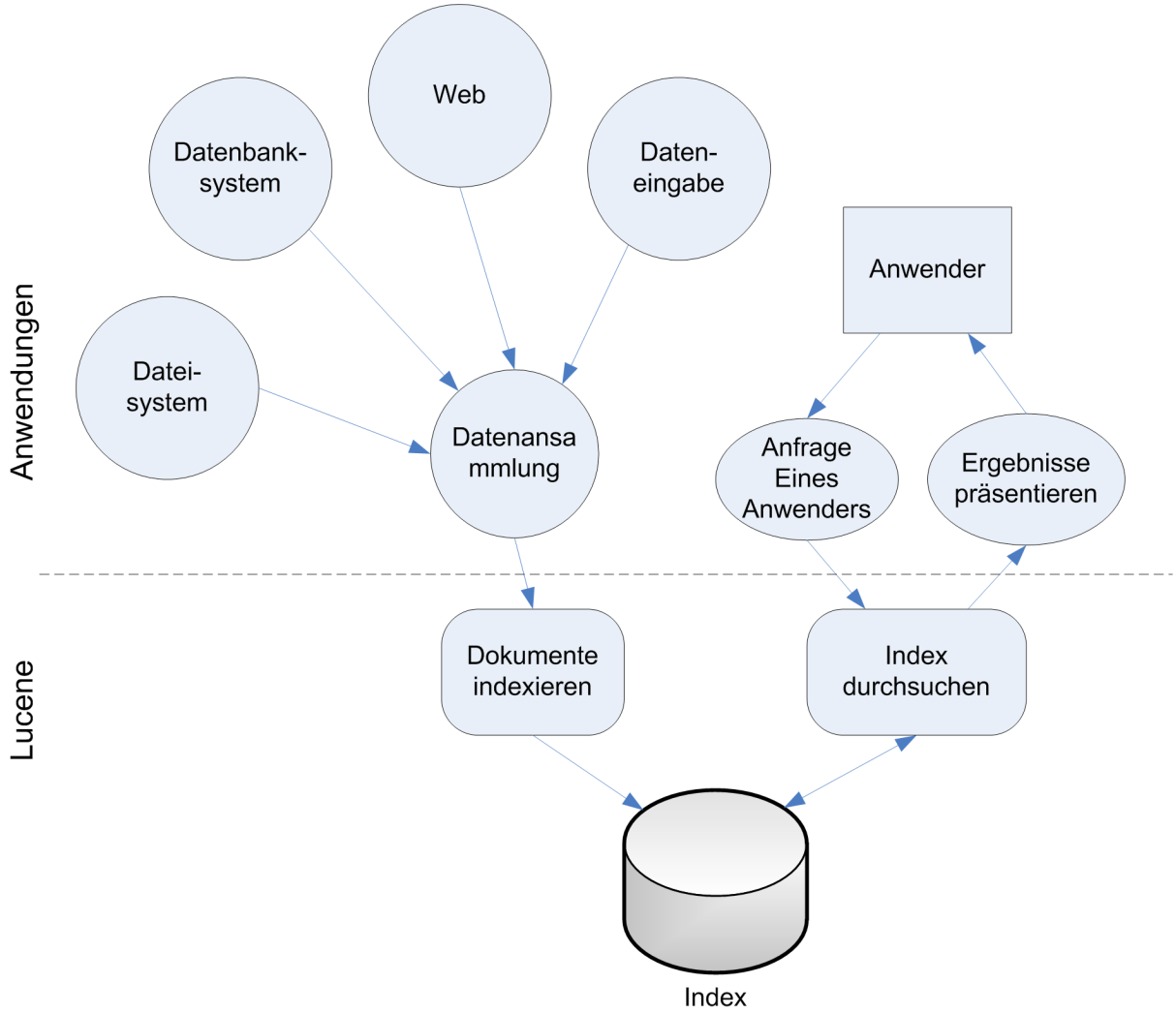

Abbildung 11: Typische Anwendungsintegration mit Lucene

Lucene kann jede beliebige Datenquelle zum Indexieren der Dokumente nutzen. Die Aufgabe des Entwicklers ist es, die Verbindung zu den Datenquellen zu implementieren. Der Hauptteil der Implementierung liegt aber in der Entwicklung von Parsern. Diese wandeln die jeweiligen Dokumente in Text um. Dieser Prozess ist notwendig, weil Lucene nur Dokumente indexieren kann, die in Textform vorliegen. So müssen beispielsweise PDF-, HTML- oder Microsoft Word- Dokumente in reinen Text umgewandelt werden. Wie in Abbildung 11 zu sehen ist, wird Lucene in zwei Aufgabenbereiche eingeteilt: das Indexieren von Dokumenten und das Durchsuchen des Index. Mitunter sind die Lucene Klassen gemäß dieser Aufgabenbereiche in Packages aufgeteilt. Im Folgenden Abschnitt werden die Klassen zum Indexieren, Analysieren und Suchen beschrieben.

3.1.2 API Klassen zur Indexierung

In diesem Abschnitt werden die Klassen der Apache Lucene API vorgestellt, die für die Indexierung von Dokumenten wichtig sind; diese sind: IndexWriter, Directory, Analyzer, Document und Field.

IndexWriter

IndexWriter ist die zentrale Komponente beim Prozess der Indexierung. Mit dieser Klasse erstellt man einen Index und fügt Dokumente zum diesem hinzu. Diese Klasse dient nur zum Beschreiben. Zum Lesen oder Durchsuchen werden IndexReader oder IndexSearcher-Klassen benutzt. Der Source Code zeigt wie ein Index mit IndexWriter erstellt werden kann.

```
File indexDir = new File(„/data/index/“);
Analyzer analyzer = new StandardAnalyzer();
IndexWriter iwriter = new IndexWriter(indexDir, analyzer, true);
..
iwriter.optimize();
iwriter.close();
```

Directory

Directory ist eine Klasse, die den Speicherort des Index repräsentiert. Es handelt sich dabei um eine abstrakte Klasse. Der Speicherort kann beliebig gewählt werden. So kann ein Index nicht nur auf dem Dateisystem-, sondern auch in einer Datenbank oder im Arbeitsspeicher abgelegt werden. Lucene bietet zwei konkrete Unterklassen für das Dateisystem (FSDirectory) und den Arbeitsspeicher (RAMDirectory) an. Die FSDirectory Klasse speichert den Index in einem angegebenen Verzeichnis ab. Die Klasse RAMDirectory speichert den Index im Arbeitsspeicher ab, dabei wird der Speicher nach Beenden eines Programms gelöscht. Diese Form wird beispielsweise für Testzwecke oder für kleine Indexe angewendet. Der folgende Programmausschnitt zeigt, wie ein Index in den Arbeitsspeicher abgelegt werden kann.

```
Directory ramDir = new RAMDirectory();
Analyzer analyzer = new StandardAnalyzer();
IndexWriter iwriter = new IndexWriter(ramDir, analyzer, true);
..
```

Analyzer

Analyzer sind eine Art Filter, die beim zu indexierenden Text angewendet werden. Bevor eine Textdatei zum Index hinzugefügt wird, muss diese erstmal in Tokens, kleinste Terme im Text, zerlegt werden, um anschließend analysiert zu werden. Die Algorithmen im Analyzer entscheiden, ob ein Term zum Index hinzugefügt oder eliminiert wird.

Bei der Analyzer-Klasse handelt es sich um eine abstrakte Klasse. Lucene bietet viele konkrete Unterklassen an. Beispielsweise gibt es Unterklassen, die Stoppwörter eliminieren oder alle Wörter in Kleinbuchstaben umwandeln. Die Wahl der richtigen Analyzer ist für eine zu programmierende Suchmaschine entscheidend. Deshalb werden die von Lucene zur Verfügung gestellten Analyzer-Klassen im Abschnitt 3.1.4 näher erläutert.

Document

Ein Document repräsentiert eine Sammlung von Feldern. Ein Lucene-Document kann als ein virtuelles Dokument betrachten werden, dass beispielsweise Webseiten oder Textdateien darstellt. Die Fields eines Dokuments können beispielsweise Autor, Titel oder Datum der letzten Änderung sein und sie können beliebig benannt werden. Lucene kann nur Text in Form von Strings oder einer Reader-Klasse bearbeiten. Andere Formate, die oft als z.B PDF-Dokument oder Microsoft Word-Dokument vorliegen, müssen erst einmal in Text umgewandelt werden. Dieser Prozess wird durch Parser abgearbeitet. Lucene stellt selbst keine fertigen Klassen für die Konvertierung der Dokumente zur Verfügung. Diese müssen vom Entwickler implementiert werden. Abbildung 12 zeigt, wie andere Formate in den Index aufgenommen werden können.

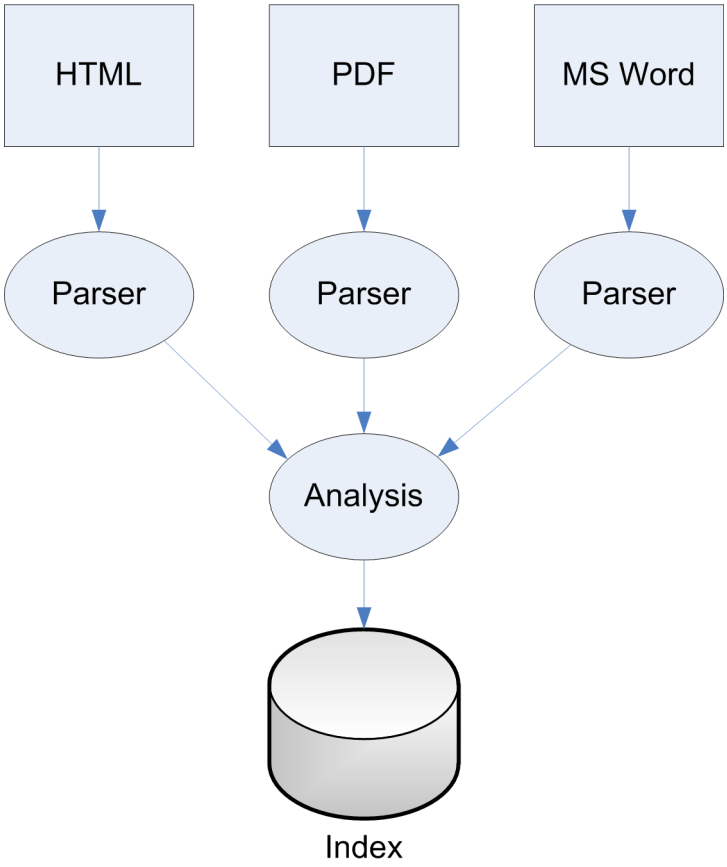

Abbildung 12: Indexing mit Lucene

Field

Jedes Lucene-Dokument hat mindestens ein oder aber mehrere Felder. Diese werden mit der Klasse „Field“ erzeugt. Jedes Feld wird zumindest mit zwei Parametern, einem Namen und einem Wert erstellt. Der Name eines Feldes kann frei gewählt werden und soll dessen Wert beschreiben. Ein Wert entspricht dem konkreten Text oder Datum eines Dokuments. Damit kann man bestimmte Abschnitte des Originaldokuments getrennt voneinander in Felder ablegen. Beispielsweise kann man ein Feld erzeugen, das den Titel eines Dokuments aufnehmen soll. Dabei könnte der Name des Feldes „Titel“ und der Wert eines Dokuments „Diplomarbeit zum Thema Volltextsuchmaschine“ lauten. Dieses Beispiel würde wie folgt als Programmcode aussehen:

```
Directory ramDir = new RAMDirectory();

Analyzer analyzer = new StandardAnalyzer();
```

```
IndexWriter iwriter = new IndexWriter(ramDir, analyzer, true);
Document doc = new Document();
String titleString = new String(„Diplomarbeit zum Thema Volltextsuchmaschine“);
String bodyString = new String(„Hier folgt der Inhalt“);
Field titleField = new Field(„titel“, titleString,
Field.Store.YES, Field.Index.UN_TOKENIZED);
Field bodyField = new Field(„content“, bodyString,
Field.Store.YES, Field.Index.TOKENIZED);
doc.add(titleField);
doc.add(bodyField);
iwriter.addDocument(doc);
iwriter.optimize();
iwriter.close();
```

Mit den folgenden Parametern kann bestimmt werden, wie ein Feld im Index abgespeichert werden soll:

Der **Field.Index** Parameter bestimmt, wie ein Feld indexiert werden soll. Es kann folgende Werte annehmen:

NO bedeutet, das Feld wird nicht in den Index indexieren. (bei dem Parameter Field.Store.YES würde das Feld nicht mehr durchsuchbar gemacht werden, aber man könnte die Originalform dieses Feldes abrufen)

TOKENIZED bedeutet, das Feld wird indexiert, damit es durchsucht werden kann. Dabei durchläuft der Originaltext einen Analyzer. Der gesamte Text wird in Terme zerlegt.

UN_TOKENIZED bedeutet, das Feld wird ohne Anwendung eines Analyzer indexiert.

Der **Field.Store** Parameter bestimmt, wie ein Originaltext eines Feldes abgespeichert werden soll. Es kann folgende Werte annehmen:

COMPRESS bedeutet, dass der original Feldwert komprimiert in den Index abgespeichert wird.

NO bedeutet, dass der Feldwert nicht abgespeichert wird.

YES bedeutet, dass der original Feldwert in den Index abgespeichert wird.

3.1.3 API Klassen zur Suche

In diesem Abschnitt werden Klassen der Lucene API vorgestellt, die für die Suche im Index wichtig sind. Folgende Klassen werden gebraucht, um die Suchoperationen auszuführen: IndexSearcher, Term, Query, TermQuery und Hits.

IndexSearcher

IndexSearcher ist die zentrale Klasse, die auf den Index verweist, um anschließend suchen zu können. Dabei benötigt die Klasse mindestens einen Parameter. Dieser Parameter ist die Stelle, an der sich der Index befindet.

```
IndexSearcher is = new IndexSearcher(„/data/index/“);
Query q = new TermQuery(new Term(„contents“, „lucene“));
Hits hits = is.search(q);
```

Term

Term repräsentiert ein Wort aus einem Text. Diese Klasse ist ähnlich der Field-Klasse, die beim Indexieren in den Index abgespeichert worden ist. Genau wie das Field, enthält Term zwei Parameter, die aus dem Namen eines Fields und dessen Wert bestehen. Dabei wird der Name eines im Index abgespeicherten Fields und einen Wert, der gesucht werden soll übergeben. Folgendes Beispiel erzeugt einen Term:

```
Query q = new TermQuery(new Term(„contents“, „lucene“));
Hits hits = is.search(q);
```

Mit diesem Ausschnitt des Programmcodes soll Lucene alle Dokumente finden, die das Wort „lucene“ im Field „contents“ beinhalten.

Query

Query ist eine abstrakte Klasse, die eine Anfrage repräsentiert. Die oben schon erwähnte konkrete Klasse TermQuery ist eine von vielen Unterklassen von Query. Abbildung 13 zeigt alle Query-Klassen, die mit Lucene erstellt werden können.

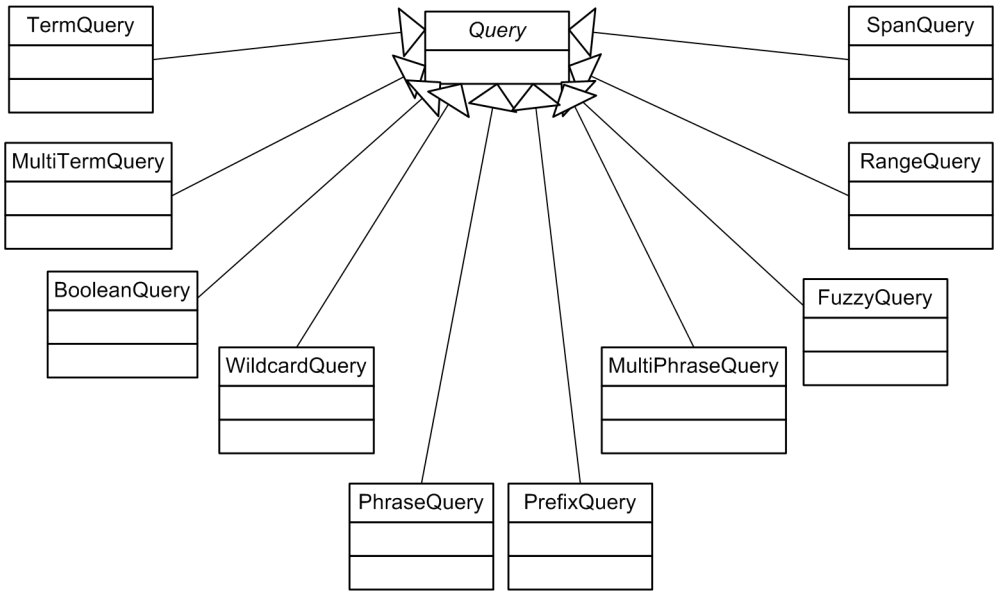

Abbildung 13: UML Diagramm der Query-Klassen

TermQuery

Bei der TermQuery handelt es sich um eine einfache Abfrage, die nur im angegebenen Field-Namen im Index sucht. Der Programmcode zeigt, wie ein TermQuery erzeugt werden kann:

```
Query q = new TermQuery(new Term(„contents“, „lucene“));
```

Hits

Die Klasse Hits repräsentiert nach Abruf einer Anfrage eine Sammlung von Ergebnissen.

Dabei handelt es sich nicht um die Inhalte aus dem Index, sondern nur um einen Verweis auf die Stellen im Index.

3.1.4 API Klassen der Analyzer

Dieser Abschnitt widmet sich dem Thema der Textanalyse. Bevor Dokumente indexiert werden können, durchlaufen sie mehrere Filter. Dieser Prozess wandelt Originaltext in Indexterme um. Indexterme sind Wörter, nach denen bei einer Anfrage im Index gesucht wird. Sie repräsentieren ein oder mehrere Dokumente. Bei diesem Prozess werden Techniken eingesetzt, wie z.B. Lemmatisierung, Eliminierung von Stoppwörtern und Stammwortreduzierung. Diese Vorgänge werden von einem Analyzer realisiert. Ein Analyzer benötigt als Eingabe einen String oder einen Reader und liefert einen sogenannten TokenStream. Abbildung 14 zeigt die Funktionsweise eines Analyzers:

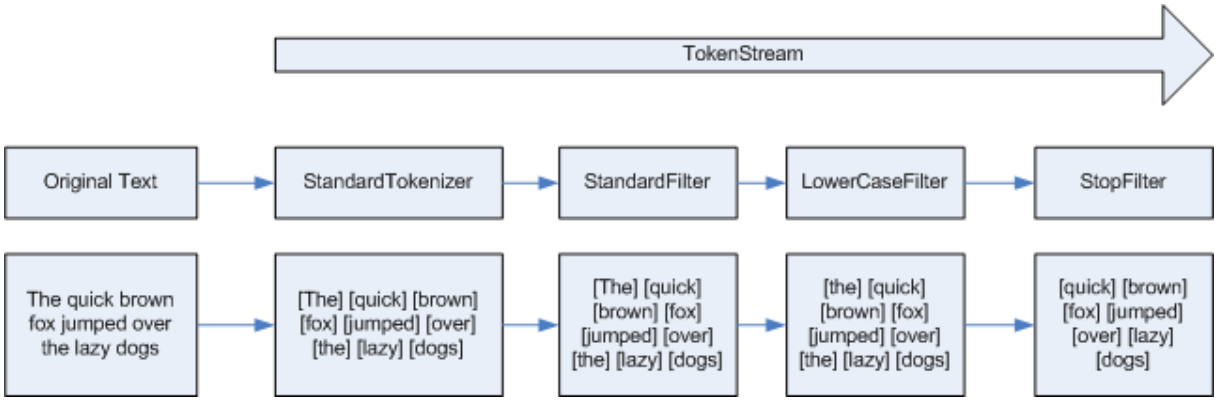

Abbildung 14: Funktionsweise und Struktur eines Analyzers

Ein Tokenizer wandelt einen Originaltext in Token um. Ein Token ist jedes einzelne Vorkommen eines Wortes in einem Text. Dabei wird jedes Wort, das durch Bindestriche getrennt ist, als ein Token erkannt. Bei Sonderzeichen wie beispielsweise @-Zeichen oder Bindestrichen wird es schwierig zu entscheiden, ob ein Wort getrennt oder zusammengesetzt betrachtet werden soll. Das kann durch die Wahl eines passenden Tokenizers erfolgen. Die vom Tokenizer erstellten Token bilden einen TokenStream, der durch TokenFilter weitergereicht wird. In der Abbildung durchlaufen die Token einen LowerCaseFilter, der dafür sorgt, dass die Token alle in Kleinbuchstaben umgewandelt werden. Als Nächstes werden die Token mit dem StopFilter bearbeitet, um Stoppwörter zu eliminieren. Ein

Analyzer kann selbst entwickelt werden oder aus der Sammlung der Lucene Analyzer ausgewählt werden. Tabelle 2 zeigt die Lucene Analyzer-Klassen und erläutert die Funktionsweise.

Analyzer Name	**Beschreibung**
KeywordAnalyzer	gibt die gesamte Eingabe als ein einzelnes Token zurück; sinnvoll bei Postleitzahlen oder ID
SimpleAnalyzer	Eingabe wird an Stellen getrennt, wo keine Buchstaben vorkommen; die Token werden anschließend in Kleinbuchstaben umgewandelt
StopAnalyzer	wie SimpleAnalyzer, nur mit dem zusätzlichen StopFilter ausgestattet; der Filter entfernt Stoppwörter, dabei wird eine englische Stoppwortliste benutzt
StandardAnalyzer	die Eingabe wird an Stellen getrennt, wo Bindestriche vorkommen, Punkte werden entfernt und E-Mail Adressen werden als ganze Token erkannt; anschließend werden die Token in Kleinbuchstaben umgewandelt und es werden Stoppwörter eliminiert; dabei wird wie beim StopAnalyzer die englische Stoppwortliste benutzt
WhitespaceAnalyzer	Eingabe wird durch Leerzeichen getrennt
PerFieldAnalyzerWrapper	mit dem Wrapper kann man zu jedem Feld einen Analyzer definieren

Tabelle 2: Lucene Analyzer

Die von Lucene zur Verfügung gestellten Analyzer-Klassen sind an die englische Sprache angepasst, weil sie beispielsweise die englische Stoppwortliste nutzen. Es gibt weitere Analyzer-Klassen, die sprachspezifisch sind. Diese werden für folgende Sprachen angeboten : portugiesisch, chinesisch, japanisch, koreanisch, tschechisch, deutsch, griechisch, französisch, niederländisch und russisch.

3.2 Eclipse Rich-Client-Platform

Die Eclipse Rich-Client-Platform ist ein Framework für die Entwicklung von Anwendungen auf Basis des Eclipse Projekts [ECLI]. Hierbei handelt es sich um ein Open-Source-Projekt der Eclipse Foundation, die eine freie und erweiterbare Java-Entwicklungsumgebung (IDE) entwickelt. Unter einem Rich-Client versteht man ein Framework, das sich auf einem Client befindet und durch Module und Plugins erweitert werden kann. Die Eclipse Java-Entwicklungsumgebung ist solch ein Framework und die jeweiligen Plugins stellen die gesamte Funktionalität der Anwendungen bereit. Dabei ist die Eclipse Rich-Client-Platform eine rudimentäre Eclipse-Version, auf der man Desktop Anwendungen entwickeln kann. Die Eclipse Rich-Client-Platform besteht nur aus einem Kern, der die notwendigen Plugins lädt. Die Plattform bietet einen flexiblen GUI-Toolkit, um vorwiegend Desktop-Anwendungen zu entwickeln. Das Einhergehen vom „Look-and-Feel“ in das jeweilige Betriebssystem sowie die hohe Performance der Plattform gegenüber anderen GUI-Toolkits, beispielsweise AWT oder Swing, machen das Framework für die Entwicklungen von Desktop Anwendungen interessant. Die im Rahmen des Gesamtprojekts festgelegte Eclipse Version 3.3 wird bei der Implementierung des Such-RCP-Plugins als Grundlage verwendet.

Als Grundlage für die Zusammenarbeit der Plugins untereinander, verwendet Eclipse den OSGi-Standard der OSGi Alliance [OSGI]. Dieser definiert eine Java-basierte Laufzeitumgebung, die es ermöglicht, im Betrieb Softwarekomponente (sogenannte Bundles) und Dienste (Services) zu laden, zu starten, zu stoppen und auszutauschen. Diese Laufzeitumgebung gibt damit die Möglichkeit, verschiedene unabhängige und modulare Anwendungen in derselben virtuellen Maschine laufen zu lassen. Dieser Standard finden in dem Java-Framework Equinox [OSGISP] Verwendung, das von der Eclipse Foundation entwickelt wurde. Equinox [EQUI] bildet das Fundament der Eclipse RCP. Zusätzlich besteht die Eclipse Rich-Client-Platform aus einer sogenannten Workbench, die den Rahmen der Benutzeroberfläche darstellt.

4 Anforderungsanalyse

Im Rahmen dieses Kapitels werden die Anforderungen an das zu entwerfende System ermittelt. Die Grundlage für diese Anforderungen bilden Ergebnisse eines Workshops für Experten, Interviews mit zukünftigen Anwendern und Gesprächen mit dem Projektleiter sowie dem Team.

4.1 Projektbeschreibung

In den Luftfahrtunternehmen werden die auf Windkanal- und Flugversuche gestützten Prozesse zur Erzeugung von aerodynamischen Daten für Lasten und Flugeigenschaften auf numerische Simulation umgestellt. Die numerische Simulation ermöglicht die Beherrschung des Entwurfsprozesses. Dadurch können die Entwicklungszeit und -risiken in der Luft- und Raumfahrt reduziert werden. Die langfristigen Ziele der numerischen Simulation sind der „Numerische Windkanal“ und die virtuelle Flugerprobung und Zertifizierung.

Die Grundlage der numerischen Strömungssimulation in der Flugzeugindustrie und im DLR bildet das am Institut für Aerodynamik und Strömungstechnik entwickelte CFD-Programm TAU (CFD = Computational Fluid Dynamics) Diese Software wird zusammen mit Hilfe eines CAD-Programms benutzt. Dabei wird vom Ingenieur ein Geometriemodell eines Flugkörpers mit einem CAD (CAD = Computer Aided Design) Software erstellt. Anschließend wird ein numerisches Netz auf der Oberfläche des Geometriemodells abgebildet. Anhand dieses Netzes kann die CFD-Software TAU die konkrete Strömung berechnen. Die erzeugten Simulationsergebnisse liegen in ASCII-Tabellen vor und können mit weiteren Werkzeugen wie z.B tau2plt bearbeitet werden. Das Werkzeug tau2plt bereitet die Ergebnisse so auf, dass diese anschließend mit einem Visualisierungswerkzeug betrachtet werden können. Die Softwarewerkzeuge müssen konfiguriert und miteinander verknüpft werden.

In der industriellen Anwendung numerischer Verfahren sind standardisierte Vorgehensweisen wichtig, um die Vergleichbarkeit von Ergebnissen unterschiedlicher Simulationen sicherzustellen. Dazu werden Best-Practice-Richtlinien erstellt. Diese sollen anhand von Anleitungen-, einen unerfahrenen oder erfahrenen Anwender, auf sinnvolle- und auf nicht anzuratende Vorgehensweisen der numerischen Verfahren hinweisen. Die Anleitungen werden von Experten der numerischen Verfahren erstellt. Dieses Wissen besteht nicht nur aus Erfahrungen der Experten, sondern auch aus einer Ansammlung von Dokumenten. Dazu soll

ein Softwaresystem entworfen und entwickelt werden, welches einen Anwender bei seiner Arbeit unterstützen soll. Deshalb ist das Ziel des Instituts für Aerodynamik und Strömungstechnik *„... den Prototypen eines „Best-Practice“ Softwaresystems für den Ströumgslöser TAU zu entwerfen, zu entwickeln, mit Inhalten zu füllen und letztendlich kontinuerlich zu pflegen und zu verbessern. Das System soll den Anwender dabei unterstützen, qualitativ hochwertige Berechnungsergebnisse zu produzieren, Unsicherheiten und Fehler im gesamten numerischen Entwurfsprozess zu minimieren und so letztendlich die Kosten (Zeit und Geld) für ein möglichst breites Spektrum von CFD-Anwendungsbereichen deutlich zu reduzieren.“*

Das Gesamtsystem besitzt die Eigenschaft eines Expertensystems und einer Wissensansammlung. Im Rahmen einer Marktverfügbarkeitsstudie von regelbasierten Expertensystemen wurden folgende elementare Anforderungen an ein Expertensystem gestellt:

- Die Arbeit mit dem Expertensystem sollte einen iterativen Prozess der Modifikation und Speisung mit Wissen unterstützen. D.h. insbesondere, dass die Spezifizierung und Modifikation von Regeln im Betrieb durch den Experten und nicht durch den Programmierer erfolgen muss.
- Das Expertensystem sollte in der Lage sein, mit „unvollständigen“ Informationen umzugehen: Häufig lassen sich nur Aussagen mit einem eingeschränkten Grad an Sicherheit machen, in der Art, dass eine Schlussfolgerung erst dann gelten wird, wenn gewisse Bedingungen erfüllt werden. Das heißt, bereits eine Teilspezifikation des Problems sollte „vernünftige“ Best Practices ausgeben, die alle noch möglichen Lösungsrichtungen, in die das Problem sich entwickeln kann, widerspiegeln.

Das Gesamtsystem wird zunächst als Prototyp konzipiert, implementiert und evaluiert. Dieser Prototyp soll als Diskussionsgrundlage für eine endgültige Implementierung dienen.

4.2 Systemübersicht und Abgrenzung

Das Gesamtsystem ist in mehrere Komponenten aufgeteilt. Dazu wurde im Team eine Architektur entworfen. Abbildung 15 zeigt die einzelnen Komponenten. Die vorliegende Arbeit befasst sich nur mit dem Teilbereich „Search Layer“. Die anderen Komponenten werden im Folgenden erläutert und sind kein direkter Bestandteil der Arbeit.

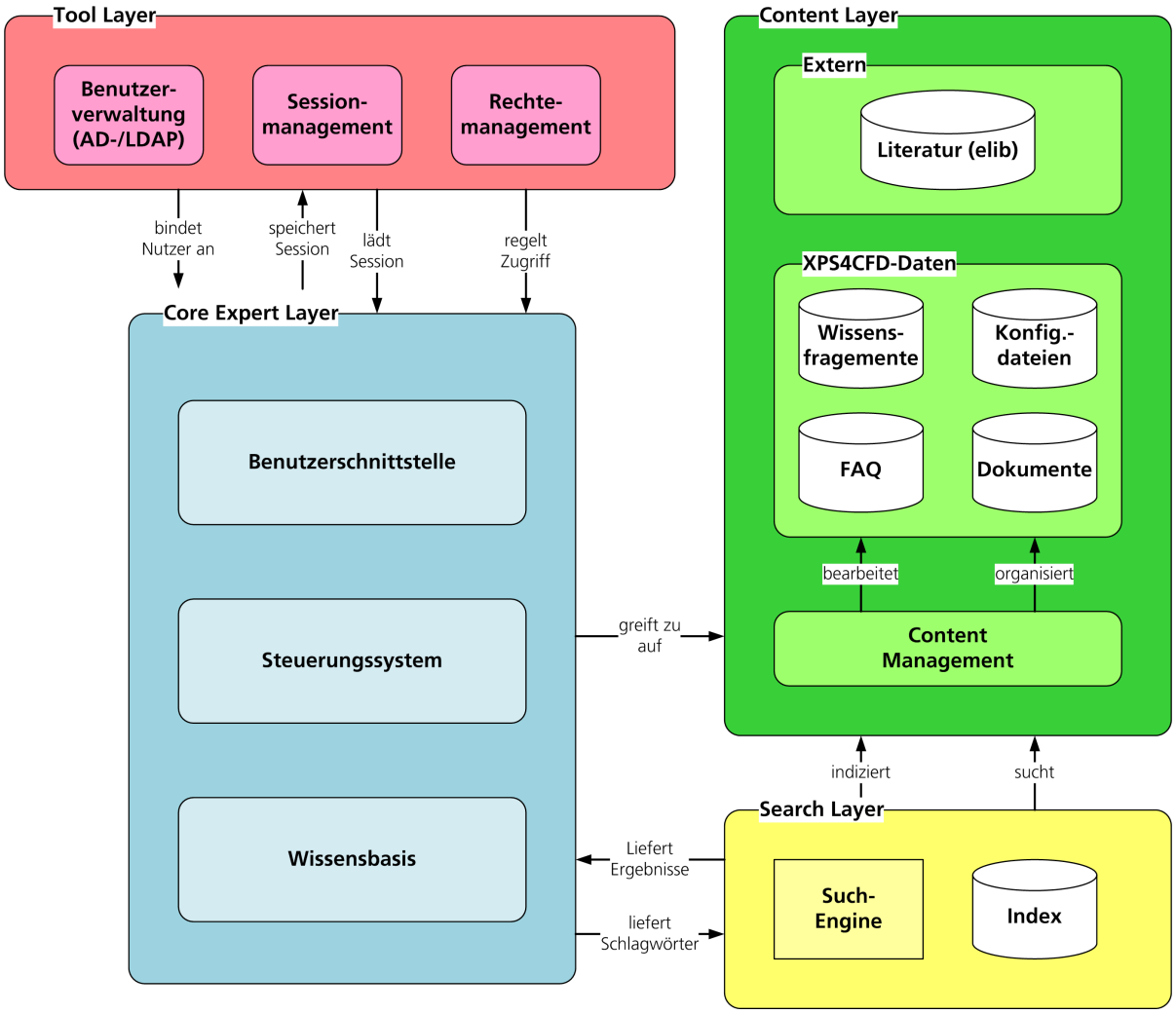

Abbildung 15: Überblick über das Gesamtsystem.

Tool Layer: Diese Komponente besteht aus Benutzerverwaltung, Session Management und Rechtemanagement. Bei der Benutzerverwaltung handelt es sich um eine Schnittstelle zum Active Directory und LDAP Service. Damit werden zentral abgelegte Benutzerprofile genutzt, um im Rechtemanagement die Berechtigung der Nutzer zu bestimmen. So kann sichergestellt werden, dass nur berechtigte Anwender das System nutzen. Das Session Management verwaltet nach Bedarf die Sessions, die im Expertensystem vom Benutzer geladen und gespeichert werden. Eine Session ist eine Sitzungsperiode, die ein Anwender ausübt, um sein Wissen zu stillen.

Content Layer: Diese Komponente stellt die Wissenssammlung des Systems dar. Die interne Wissenssammlung wird mit Hilfe des Contentmanagements verwaltet. Diese Daten sind

Wissensfragmente des Expertensystems, Konfigurationsdateien, FAQ und Dokumente. Sie liegen als strukturiertes Wissen vor, also in einer relationalen Datenbank oder als unstrukturiertes Wissen in Form von beispielsweise PDF, Word-Dokumenten. Die externe Wissenssammlung, die im DLR zum Einsatz kommt, ist die Literaturdatenbank elib. Diese verwaltet aktuell ca. 60.000 wissenschaftliche Dokumente. Das Content Layer befindet sich in Entwicklung und soll prototypisch implementiert werden.

Core Expert Layer: Das Core Expert Layer ist die zentrale Systemkomponente des Gesamtsystems. Es handelt sich hierbei um ein regelbasiertes Expertensystem. Dieses hilft dem Experten Regeln, Szenarien, Workflows und Anleitungen zu erstellen und abzuspeichern. Das Layer besteht aus der Benutzerschnittstelle, dem Steuerungssystem und einer Wissensbasis. Die Benutzerschnittstelle dient der Interaktion mit den verschiedenen Anwendern, wie z.B mit einem Experten. Das Steuerungssystem sorgt dafür, dass die vom Experten erstellten Szenarien nach definierten Regeln ablaufen. Die Wissensbasis beinhaltet die zuvor genannten Szenarien, Regeln und Workflows. Diese Komponente befindet sich in Entwicklung und soll wie das Content Layer prototypisch implementiert werden.

Search Layer: Das Search Layer soll im Rahmen dieser Arbeit als Prototyp entworfen und implementiert werden. Die Komponente besteht aus der Such-Engine und dem Index. Die Such-Engine stellt einen Suchdienst bereit, der direkt vom Anwender oder vom Steuerungssystem des Expertensystems ausgeführt werden kann. Die Suche kann mit einer einfachen, aber auch mit einer komplexen Anfrage erfolgen. Die Ergebnisse einer Suche werden vom Core Expert Layer an die Benutzerschnittstelle weitergeleitet. Die Such-Engine nutzt einen eigenen Index. Dieser beinhaltet das unstrukturierte Wissen aus dem Content Layer. Dazu werden Dokumente in den Index indiziert.

4.3 Funktionale und nicht-funktionale Anforderungen

In diesem Abschnitt werden funktionale und nicht-funktionale Anforderungen an die zu entwickelnde Suchmaschine aufgelistet. Diese sind anhand von Ergebnissen eines Workshops für Experten und zukünftige Endanwender- sowie Gesprächen mit dem Projektleiter erstellt worden.

4.3.1 Allgemeine Anforderungen

- Das Gesamtsystem soll systemunabhängig sein und sowohl unter UNIX als auch unter Windows ausführbar sein.
- Das System soll eigenständig laufen, dann jedoch möglicherweise mit beschränktem Funktionsumfang.
- Eine einfache Installation von einem Installationsmedium soll möglich sein.

4.3.2 Funktionale Anforderungen

- Die Suchmaschine soll Anfragen des Nutzers umsetzen und ihm Ergebnisse liefern.
- Szenarien des Gesamtsystems sollen mit der Suchmaschine des Search Layers kommunizieren können und die gefundenen Elemente darstellen.
- Dokumente sollen aus einer Ergebnisliste geöffnet werden.
- Die Suchmaschine soll folgende Dokumenttypen indizieren: TXT, RTF, HTML, XML, PDF, XLS, DOC, PPT und OpenOffice.

4.3.3 Nicht-funktionale Anforderungen

- Einfache Suchmaske soll dem Anwender zur Verfügung stehen.
- Zusätzlich soll eine Erweiterte Suchmaske zur Verfügung stehen, die es dem Anwender erlaubt, gezielt in bestimmten Kategorien zu suchen.
- Die Ergebnisliste soll zusätzliche Informationen, wie Autor, Titel und Dateigröße, über die angezeigten Dokumente liefern.
- Die zu entwickelnde Software soll ausschließlich mit frei verfügbarer Software programmiert werden.
- Das Indizieren von Dokumenten soll möglichst automatisch erfolgen.
- Die Dokumente, die indexiert werden, liegen alle zentral in einem Ordner vor.

4.3.4 Randbedingungen

- Die Suchmaschine wird auf Basis von Apache Lucene 2.3.2 konzipiert und aufgebaut.
- Als Programmiersprache wird Java 6 verwendet.
- Das Expertensystem wird auf Basis der Eclipse Rich Client Platform 3.3 umgesetzt.
- Die wissenschaftlichen Dokumente liegen in englischer Sprache vor.

Im Rahmen dieser Arbeit soll das in Abbildung 15 gezeigte Search Layer konzipiert und prototypisch implementiert werden. Die anderen Layer, wie das Tool Layer, Core Expert Layer und das Content Layer werden parallel zur dieser Arbeit konzipiert und entwickelt. Abbildung 15 zeigt die Verbindungen zwischen den Layern. Aufgrund dessen können bei Änderungen in den anderen Komponenten die oben genannten Anforderungen leicht variieren. Zusätzlich wurde von allen Beteiligten erkannt, dass aufgrund der zeitlichen Beschränkung eine vollständige Konzeption und prototypische Umsetzung aller Anforderungen des Gesamtsystems nicht zu erreichen ist. Deshalb ist das Ziel des Projekts ein Nachweis der Machbarkeit, sowie das Sammeln von Erfahrungen anhand eines Grundkonzepts. Der Fokus bei dem zu entwickelnden Prototypen der Suchmaschine liegt erst einmal beim indizieren der wissenschaftlichen Dokumente und einer Suchmaske für Benutzer.

4.4 Anwendungsfälle des Systems

Die folgenden Anwendungsfälle zeigen die Interaktion zwischen dem Akteur und dem Suchsystem. Unabhängig von den oben dargelegten Anforderungen und Randbedingungen können zwei wesentliche Merkmale bei dem zu entwickelnden Suchsystem festgelegt werden. Diese sind: die automatische Indexierung der wissenschaftlichen Dokumente und deren Wiederfinden (Retrieval). Bevor aber die Anwendungsfälle beschrieben werden, müssen die Akteure zunächst dargestellt werden. Bei dem Akteur Anwender handelt es sich um eine natürliche Person, die primär ein wissenschaftlicher Mitarbeiter oder eine wissenschaftliche Mitarbeiterin ist, die im CFD-Umfeld arbeitet.

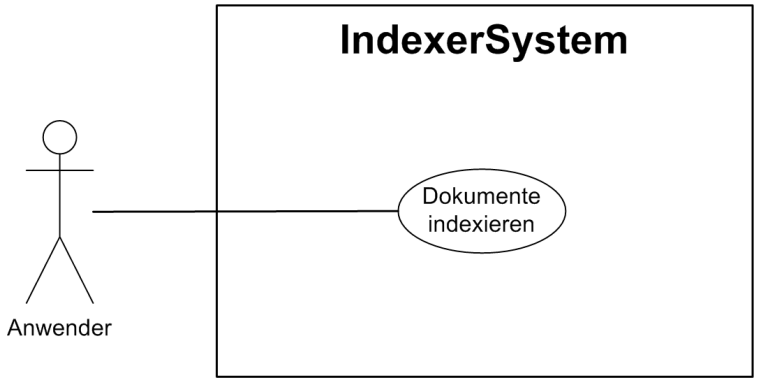

Abbildung 16: Anwendungsfalldiagramm IndexerSystem

Dokumente indexieren

Beschreibung: Der Anwender legt einen Dateiordner fest, der indiziert werden soll. Anschließend führt er den Indexer aus.

Akteure: Anwender

Vorbedingungen: Der Anwender muss am System angemeldet sein.

Nachbedingungen: Eine Übersicht der indizierten Dokumente wird angezeigt.

Ablaufschritte:

1. Der Anwender gibt die zu indexierenden Ordner, sowie den zu speichernden Indexordner an.
2. Der Anwender drückt die Return Taste, um den Indexer zu starten.
3. Das System indexiert die jeweiligen Dokumente, aus den Ordner, die der Anwender festgelegt hat.
4. Der Benutzer erhält während der Indexierung Informationen darüber, welches Dokument aktuell bearbeitet wird.

Ausnahme:

- Ungültige Eingabe des Anwenders
- Dokumente nicht verfügbar
- Index nicht verfügbar

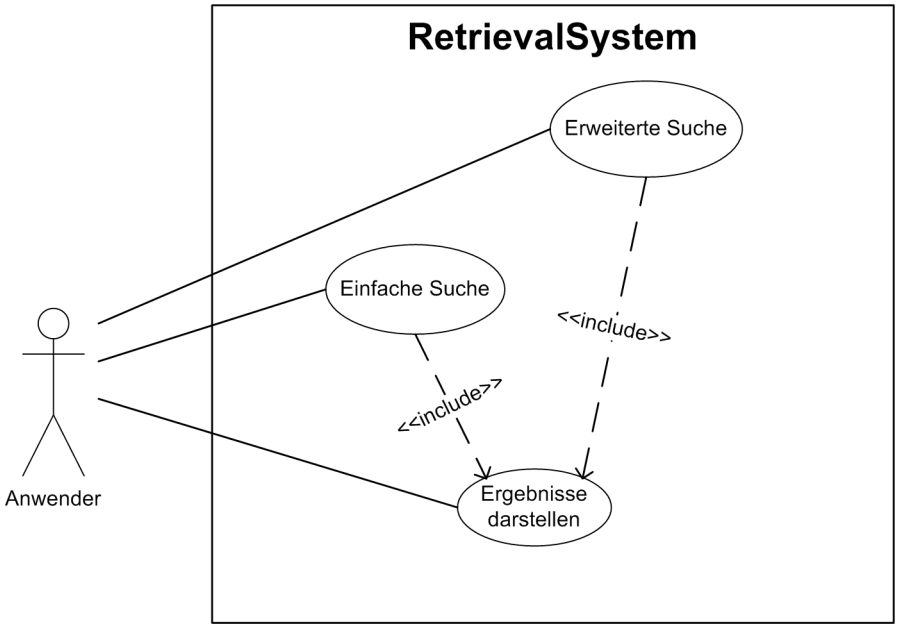

Abbildung 17: Anwendungsfalldiagramm RetrievalSystem

Anwendungsfall : Einfache Suche

Beschreibung: Der Anwender formuliert seine Anfrage und führt diese aus. Der Anwender bekommt vom System die relevanten Dokumente als Ergebnisse angezeigt.

Akteure: Anwender

Vorbedingungen: Der Anwender muss am System angemeldet sein und befindet sich in der Suchmaske.

Nachbedingungen: Die Ergebnisse der einfachen Suchanfrage werden angezeigt.

Ablaufschritte:

1. Der Anwender formuliert seine Anfrage
2. Der Anwender klickt auf den Button „Search“.
3. Das System führt die Suche aus.
4. Das System liefert relevanten Dokumente als Ergebnis der Anfrage.

Ausnahme:

- Ungültige Eingabe des Anwenders
- Index nicht verfügbar

- Keine Ergebnisse

Anwendungsfall : Erweiterte Suche

Beschreibung: Der Anwender formuliert seine Anfrage anhand von mehreren vordefinierten Feldern und führt diese aus. Er bekommt vom System die relevanten Dokumente als Ergebnisliste angezeigt.

Akteure: Anwender

Vorbedingungen: Der Anwender muss am System angemeldet sein und befindet sich in der erweiterten Suchmaske.

Nachbedingungen: Die Ergebnisse der strukturierten Suchanfrage werden angezeigt.

Ablaufschritte:

1. Der Anwender formuliert seine Anfrage in vordefinierten Feldern.
2. Der Anwender klickt den Button „Search“.
3. Das System führt die Suche aus.
4. Das System liefert relevanten Dokumente als Ergebnis der strukturierten Anfrage.

Ausnahme:

- Ungültige Eingabe des Anwenders
- Index nicht verfügbar
- Keine Ergebnisse

Anwendungsfall : Ergebnisse darstellen

Beschreibung: Der Anwender betrachtet die Ergebnisse und kann die gefundenen Dokumente öffnen.

Akteure: Anwender

Vorbedingungen: Der Anwender muss am System angemeldet sein und befindet sich in der Ergebnismaske. Ergebnisse sind vorhanden und werden angezeigt.

Nachbedingungen: keine

Ablaufschritte:

1. Der Anwender betrachtet die Ergebnisse.
2. Er klickt auf den für ihn relevanten Dokumentenlink.
3. Das Dokument öffnet sich.

Alternative Ablaufschritte:

1. Der Anwender betrachtet die Ergebnisse.
2. Die Ergebnisse sind für den Anwender nicht relevant.
3. Der Anwender formuliert seine Anfrage in der entsprechenden Suchmaske neu.

Ausnahme:

- Ungültige Eingabe des Anwenders.
- Index nicht verfügbar
- Keine Ergebnisse

5 Entwurf des Suchsystems

Im folgenden Kapitel wird ein Entwurf für das zu entwickelnde Suchsystem dargestellt. Die Grundlagen hierfür bilden die im vorherigen Kapitel definierten Anforderungen. Der Entwurf teilt sich in die zwei wesentlichen Bereiche - IndexerSystem sowie RetrievalSystem - auf. Abbildung 18 zeigt die Übersicht des Systems, die im Folgenden erläutert wird.

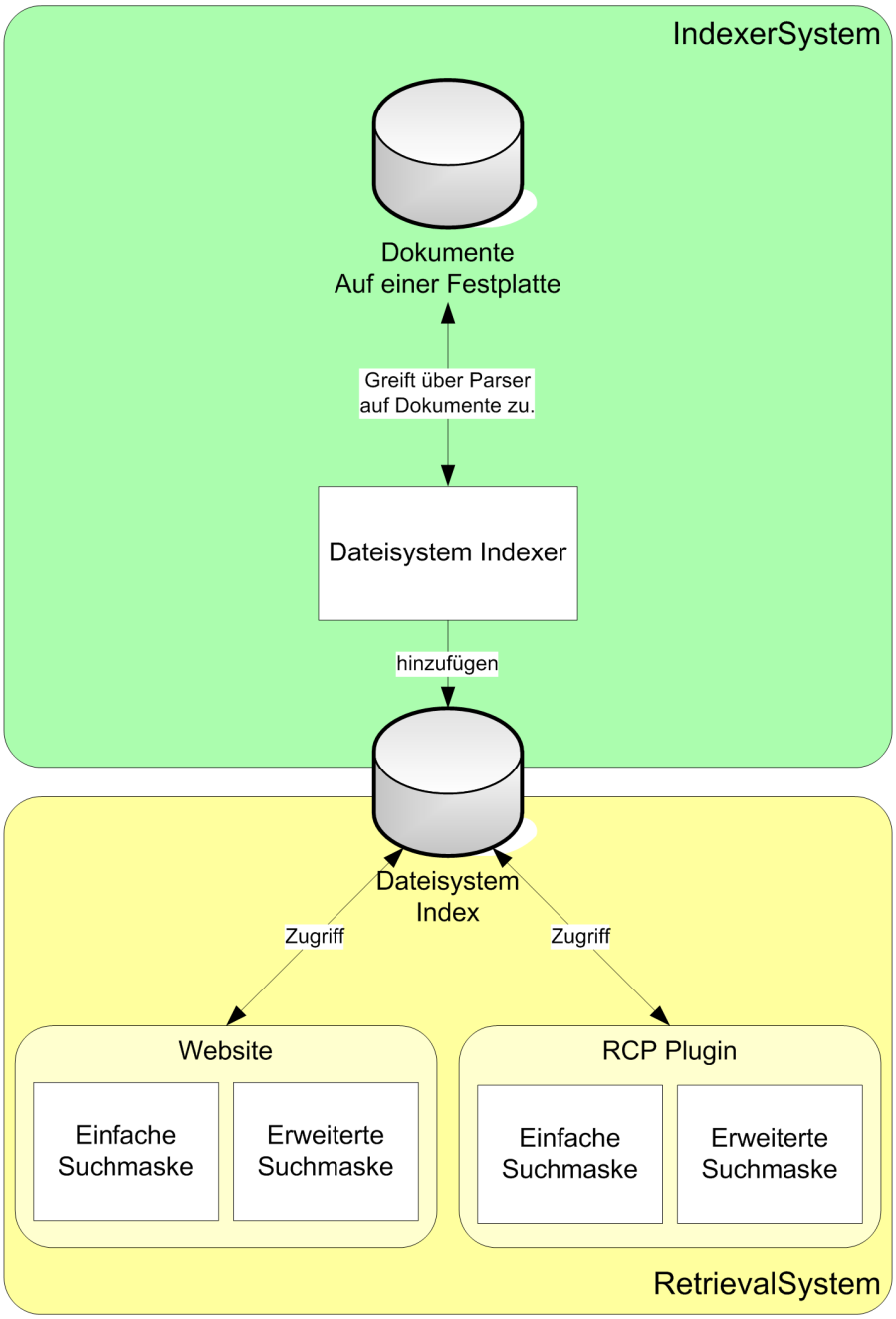

Abbildung 18: Übersicht des Suchsystems

IndexerSystem zur Erstellung des Suchindex

Das IndexerSystem hat die Aufgabe, einen Index mit allen Dokumenten zu erstellen, die für die Volltextsuche relevant sind. Dazu wird ein Dateisystem-Indexer entwickelt, der mit Hilfe von Parsern Dokumente indiziert. Laut der Anforderungsdefinition sollen folgende Dokumenttypen verarbeitet werden: TXT, RTF, HTML, XML, PDF, XLS, DOC, PPT und entsprechende Datentypen von OpenOffice. Hierzu werden Parser entwickelt, die den Inhalt der Dokumente extrahieren können und diese anschließend dem Analyzer übergeben. Dieser Prozess wurde bereits in Unterkapitel 3.1.2 erläutert. Die Sprache der Inhalte der Dokumente spielt bei der Auswahl eines Analyzers eine große Rolle. Laut der Anforderungsdefinition soll die englische Sprache benutzt werden. Der StandardAnalyzer genügt dieser Anforderung und kann zum Verarbeiten der Inhalte angewendet werden. Zunächst muss hierzu der Indexer auf die Dokumente zugreifen können, dazu werden vom Anwender entsprechende Ordner angegeben. Diese werden rekursiv nach Dateien durchsucht. Jede in das oben angegebene Schema passende Datei wird überprüft, ob sie geparst werden können. Trifft dies zu, so werden die Inhalte sowie die Metadaten dem Index hinzugefügt.

RetrievalSystem zur Abfrageverarbeitung

Das RetrievalSystem stellt die Schnittstelle zum Benutzer dar. Dazu werden zwei unterschiedliche grafische Benutzeroberflächen entwickelt. Als Erstes wird eine Webseite mit einer einfachen und einer erweiterten Suchmaske entwickelt, die den Zugriff auf die zuvor erstellten Indexe bereitstellt. Die beiden unterschiedlichen Suchmasken sollen den Anwender dabei unterstützen, seine Anfragen zu spezifizieren. Die einfache Suchmaske stellt nur ein Eingabefeld zur Verfügung, wohingegen die erweiterte Suchmaske dem Anwender die Möglichkeit gibt, in bestimmten Feldern der Indexdokumente zu suchen. Dazu werden Eingabefelder eingeblendet, die den Namen der Metadaten der Indexdokumente entsprechen.

Bei der zweiten Benutzeroberfläche handelt es sich um ein integriertes RCP-Plugin für das Expertensystem. Dieser besteht aus einer View, die ebenfalls je zwei Suchmasken enthält. Zusätzlich bietet das Plugin einen OSGI-Service, der von anderen Komponenten des Expertensystems genutzt werden kann. Dieser Service wird der Anforderung „Szenarien des Gesamtsystems sollen mit der Suchmaschine des Search-Layers kommunizieren können und die gefundenen Elemente darstellen können“ gerecht. Der Dienst nimmt Anfragen von Szenarien oder anderen OSGi-Bundles entgegen und stellt die dazu passenden Ergebnisse in

der Plugin-View dar. Auch hier werden dem Anwender über das RCP-Plugin sowohl eine einfache, als auch eine erweiterte Suchmaske zur Verfügung gestellt, diese sind mit den Webseiten-Suchmasken identisch.

5.1 Gliederung des Entwurfs

Abbildung 18 auf Seite 49 zeigt die zu entwickelnde Systemarchitektur, die in drei in sich geschlossene Aufgabenfelder aufgeteilt wird. Jedes Aufgabenfeld wird als Entwurf detailliert in Unterkapiteln beschrieben. Diese sind :

- Dateisystem-Indexer
- Webseite als Benutzeroberfläche für den Anwender im Firmen-Intranet
- Eclipse RCP-Plugin als integrierte Benutzeroberfläche für den Anwender des Expertensystems

5.2 Entwurf des Dateisystem-Indexers

Der Dateisystem-Indexer hat die Aufgabe Inhalte und Metadaten eines Dokuments zu extrahieren und diese anschließend in einem Index geeignet für schnellen Zugriff abzu speichern. Diese kurze Erläuterung beschreibt nur die allgemeine Funktionsweise eines Indexers. Zusätzlich wurden im Kapitel 4.3 Anforderungen definiert, die für den Indexer relevanten Anforderungen im Folgenden aufgelistet werden:

1. Die Suchmaschine soll folgende Dokumenttypen indizieren: TXT, RTF, HTML, XML, PDF, XLS, DOC, PPT und OpenOffice-Datentypen.
2. Die Ergebnisliste soll zusätzliche Informationen wie Autor, Titel und Dateigröße über die angezeigten Dokumente liefern.
3. Das Indizieren von Dokumenten soll möglichst automatisch erfolgen.
4. Die Dokumente, die indiziert werden liegen alle zentral in einem Ordner vor.
5. Das Gesamtsystem soll System unabhängig und sowohl unter UNIX als auch unter Windows lauffähig sein.
6. Die wissenschaftlichen Dokumente liegen in englischer Sprache vor.

Der Rest dieses Abschnitts beschreibt nun den Entwurf des Dateisystem-Indexers unter Beachtung der genannten Anforderungen. Laut dem ersten Punkt der Anforderungsliste sollen verschiedene Dokumenttypen indiziert werden. Diese können nicht direkt von den Indexierungs-Klassen der Lucene API verarbeitet werden. Deshalb ist zu jedem Dokumenttyp ein zu Parser erstellen. Ein Parser hat die Funktion, Inhalt sowie Meta-Informationen aus einem Dokument zu extrahieren. Diese Daten werden in Feldern abgespeichert (in Lucene Fields genannt). Alle Felder von einem Dokument ergeben ein Indexdokument, das dem Index hinzugefügt wird. Da im Verlauf der Zeit neue Dokumenttypen hinzukommen können, ist der Entwurf so zu realisieren, dass der Indexer mit neuen Parsern erweitert werden kann. Daher wird bei der Entwicklung der Parser das Factory-Muster angewendet, dessen Prinzip lautet [ENT06]:

„Stützen Sie sich auf Abstraktionen. Stützen Sie sich nicht auf konkrete Klassen“

Abbildung 19 zeigt das Klassendiagramm der Parser-Klassen und der Factory-Klassen.

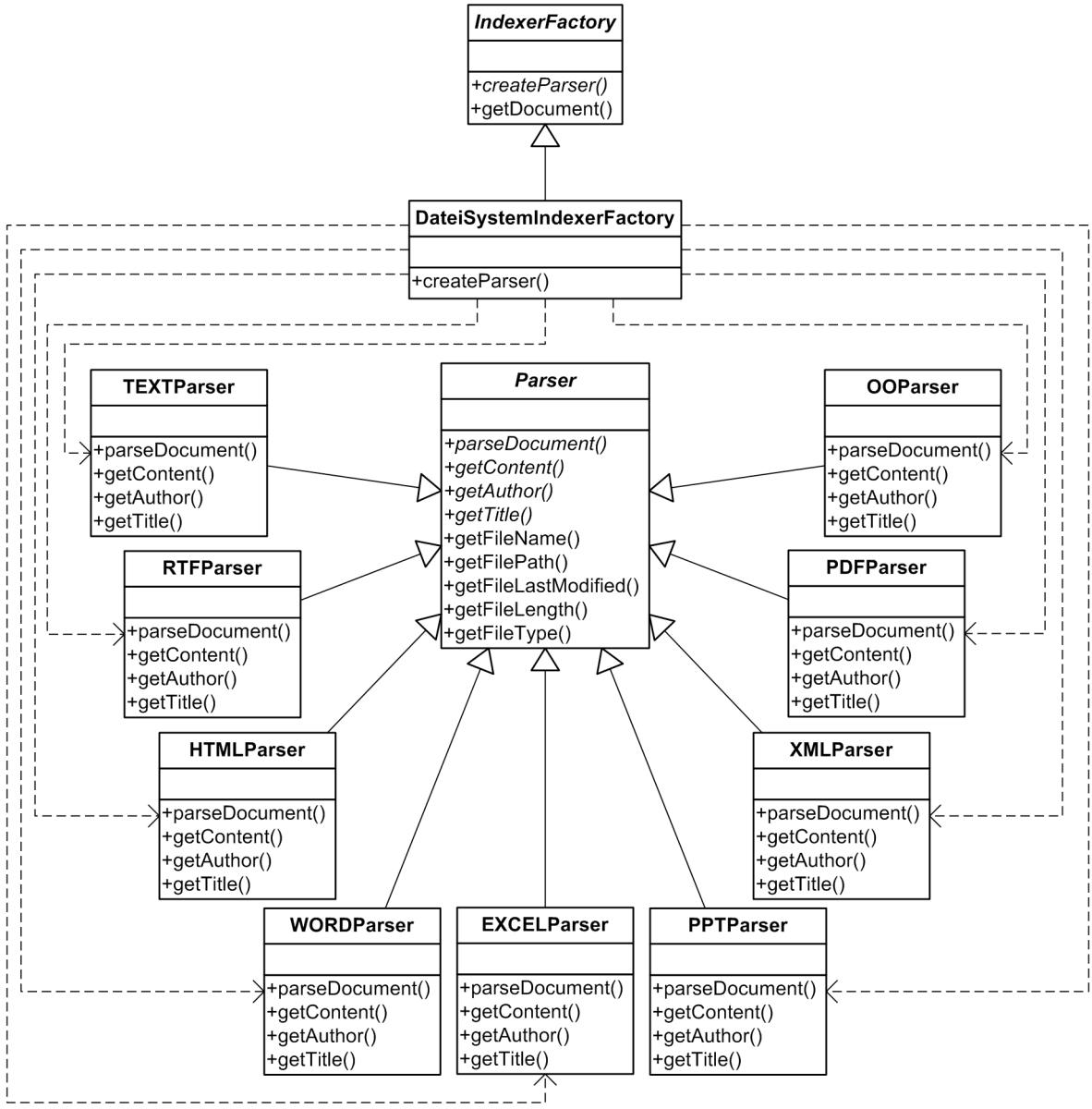

Abbildung 19: UML Klassendiagramm zeigt Parser und wie diese erstellt werden

Die zweite Anforderung lautet, dass die Ergebnisliste zusätzliche Informationen wie Autor, Titel und Dateigröße anzeigen soll. Diese Informationen können in der Parser-Klasse aus einem Dokument extrahiert werden. Manche Informationen können aber nicht immer ermittelt werden, weil etwa in einer XML-Datei einen Titel nicht bestimmt werden kann. In einer PDF-Datei lassen sich Titel, Autor und weitere Meta-Informationen abfragen. Die Verfügbarkeit gewisser Meta-Informationen ist also abhängig vom Dokumenttyp. Es gibt auch Meta-Informationen, die immer bestimmt werden können. So kann z.B der Name, der Pfad, das Datum der letzten Änderung im System, die Länge sowie der Typ einer Datei bestimmt werden. Tabelle 3 zeigt die Übersicht der genannten Methoden:

Parser-abhängige Methoden für Meta-Informationen
getContent()
getAuthor()
getTitle()
Parser-unabhängige Methoden für Meta-Informationen
getFileName()
getFilePath()
getFileLastModified()
getFileLength()
getFileType()

Tabelle 3: Parser-Methoden zur Gewinnung von Meta-Informationen.

Betrachtet man die Anforderungen drei und fünf, so kann der Indexer als ein Softwaretool entworfen werden, das regelmäßig und automatisch vom Betriebssystem gestartet wird. Das kann mit sogenannten „Geplanten Tasks“ (unter Windows-basierten Betriebssystemen) oder mit Hilfe von „cronjobs“ (unter Unix üblich) erfolgen. Die Systemunabhängigkeit des Indexers ist durch den Gebrauch der Programmiersprache Java gegeben. Die Randbedingung sechs besagt, dass die wissenschaftlichen Dokumente sämtlich in der englischen Sprache vorliegen, dem entsprechend beschränkt sich der Indexer nur auf diese Sprache. Dazu kann der StandardAnalyzer vom Lucene verwendet werden. Der vierte Punkt findet Eingang in die Konzeption der Funktionsweise des Dateisystem-Indexers. Abbildung 20 zeigt einen Programmablaufplan des Dateisystem-Indexers.

Folgende Schritte werden im Programmablaufplan beschrieben:

1. Zusammen mit Parametern startet der Anwender den Indexer. Bei den Parametern handelt es sich um Ordner, die indiziert werden sollen.
2. Als Nächstes wird eine Liste der Dateien erstellt, die in diesem Ordner vorhanden sind. Der Einfachheit halber sei diese Liste hier „Ordnerliste“ genannt.

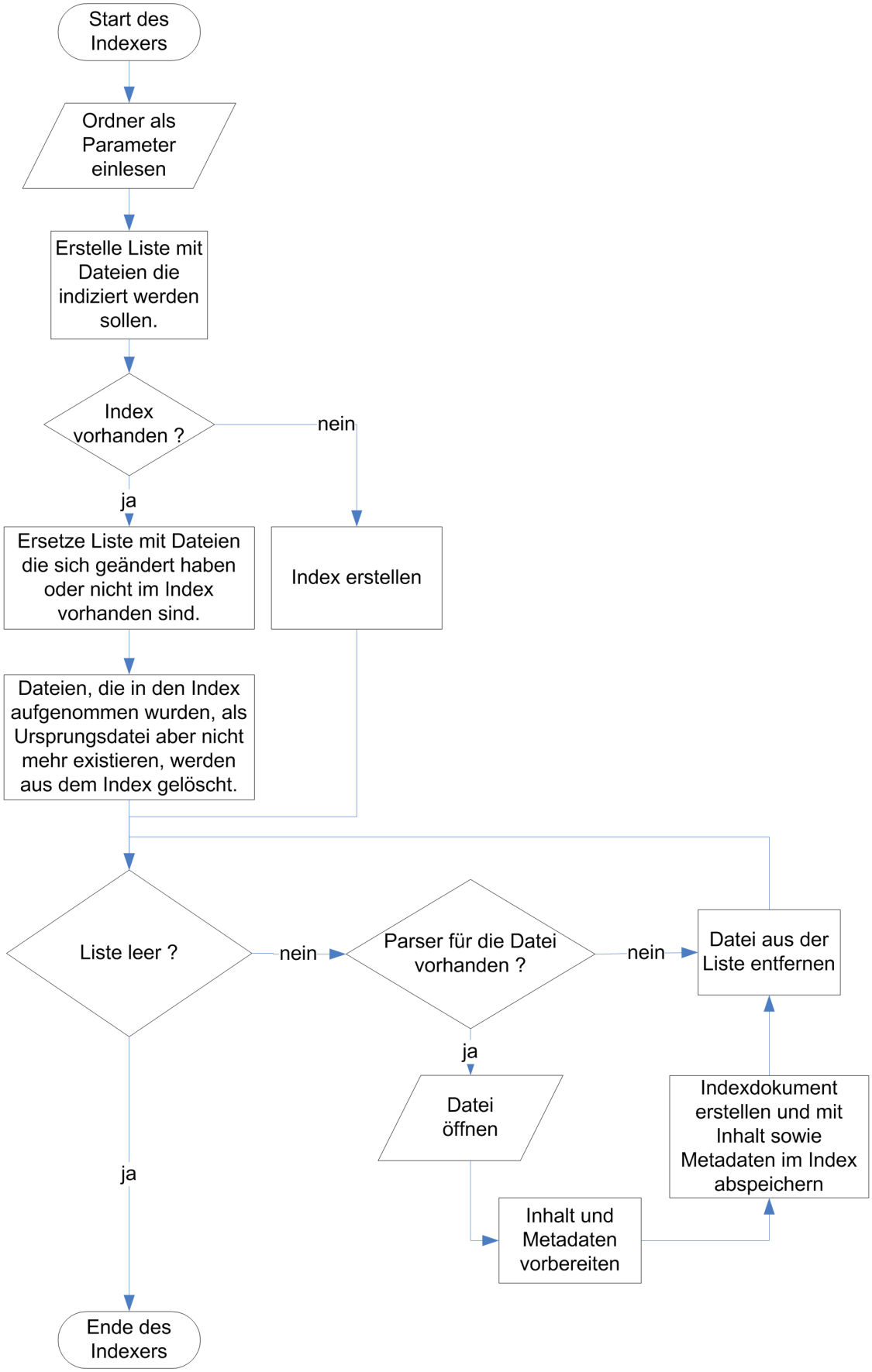

Abbildung 20: Programmablaufplan des Dateisystem-Indexers

3. Es wird überprüft, ob ein Index bereits vorhanden ist. Trifft dies nicht zu, so wird ein neuer Index erstellt, um Dokumente hinzufügen zu können. Ist der Index bereits vorhanden, so wird er nicht überschrieben, sondern nur aktualisiert.
4. Wenn der Index aktualisiert werden soll, so wird die bereits erstellte Ordnerliste mit neuen Werten überschrieben. Dabei werden Dateien, die sich gegenüber dem entsprechenden Indexeintrag verändert haben, eingetragen und es werden Dateien eingetragen, die noch nicht im Index vorhanden sind.
5. Dateien, die in den Index aufgenommen wurden, jetzt aber nicht mehr als Ursprungsdatei existieren, werden aus dem Index gelöscht.
6. Ist die Ordnerliste leer, so wird der Indexer beendet. Befinden sich noch weitere Einträge darin, so werden diese in einer Schleife abgearbeitet.
7. Für jede Datei wird überprüft, ob ein Parser vorhanden ist. Existiert kein Parser, so wird die Datei ignoriert und aus der Ordnerliste entfernt. Anschließend wird mit Schritt 6 fortgefahren. Bei Existenz eines Parser wird dieser benutzt, um die Datei zu öffnen.
8. Die mit Hilfe der Parser aus einer Datei extrahierten Meta-Informationen und der Inhalt werden in Feldern eines Indexdokuments abgespeichert.
9. Das in Schritt 8 erstellte Indexdokument wird zum Index hinzugefügt. Die aktuell bearbeitete Datei wird aus der Ordnerliste gelöscht. Der Ablauf wird mit Schritt 6 fortgesetzt.

5.3 Entwurf der Suchoberfläche als Webseite

Die Suchoberfläche stellt die Schnittstelle zum Anwender dar. Diese teilt sich laut der Anforderungen in einfache und erweiterte Suchmaske ein. Beide sollen intuitiv und effizient zu bedienen sein. Zusätzlich sollen die Suchergebnisse übersichtlich präsentiert werden. Das Design ist so konzipiert, das es sich an Suchmasken üblicher Suchmaschinen im Internet orientiert. Im Folgenden werden für den Entwurf relevante Anforderungen an die Weboberfläche aufgelistet:

1. Die Suchmaschine soll Anfragen des Nutzers umsetzen und ihm Ergebnisse liefern.
2. Eine einfache Suchmaske soll dem Anwender zur Verfügung stehen.
3. Zusätzlich soll eine erweiterte Suchmaske zur Verfügung stehen, die es dem Anwender erlaubt, gezielt in bestimmten Kategorien zu suchen.
4. Aus einer Ergebnisliste sollen Dokumente geöffnet werden können.
5. Die Ergebnisliste soll zusätzliche Meta-Informationen wie Autor, Titel und Dateigröße über die angezeigten Dokumente darstellen.

Der erste Punkt stellt eine grundsätzliche Anforderung an die Funktionsweise einer Suchmaschine. Deren Funktionen werden von der Lucene API zur Verfügung gestellt. Der Entwickler muss sich nicht um die Implementierung der Retrieval Technologien kümmern. Er kann die Klassen der Lucene API für seine Zwecke nutzen. Die weiteren Punkte in der Anforderungsliste teilen sich in drei Bereiche auf. Der erste Bereich ergibt sich aus der Anforderung an eine einfachen Suchmaske, die dem Anwender zur Verfügung stehen soll. Hierbei besteht die einfache Suchmaske aus einem Eingabefeld, einem Such-Button und einem Link zu der erweiterten Suchmaske, die weiter unten beschrieben wird. Der Anwender kann seine Anfrage im Eingabefeld formulieren. Beim Klicken auf den Such-Button wird der Suchprozess mit der formulierten Anfrage gestartet. Nach erfolgreichem Beenden des Suchprozesses werden Ergebnisse angezeigt. Der Anwender kann mit Klicken auf den „Advanced Search“ Link, die Suchmaske wechseln. Abbildung 21 zeigt den Entwurf der einfachen Suchmaske.

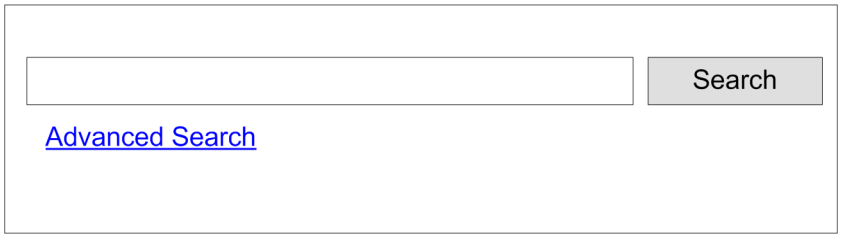

Abbildung 21: Entwurf der einfachen Suchmaske

Der zweite Bereich ergibt sich aus der Anforderung an die erweiterte Suchmaske, die dem Anwender erlaubt, in bestimmten Kategorien gezielt zu suchen. Sie besteht aus mehreren

Eingabefeldern, die mit selbst definierten Operatoren wie „Must match“, „Must not match“ oder „optional“ miteinander verknüpft werden. Hierbei handelt es sich um Felder wie Titel, Autor, Inhalt, Dateityp sowie Datum des Dokuments. Mit der Verknüpfung dieser Felder entsteht eine strukturierte Anfrage, die mit dem Such-Button gestartet werden kann. Zusätzlich zu den Feldern und dem Such-Button kann der Anwender auf einen Link klicken, der ihn in die einfache Suchmaske führt. Dieser Entwurf orientiert sich an Eingabemasken gängiger Web-Suchmaschinen. Abbildung 22 zeigt den Entwurf der erweiterten Suchmaske.

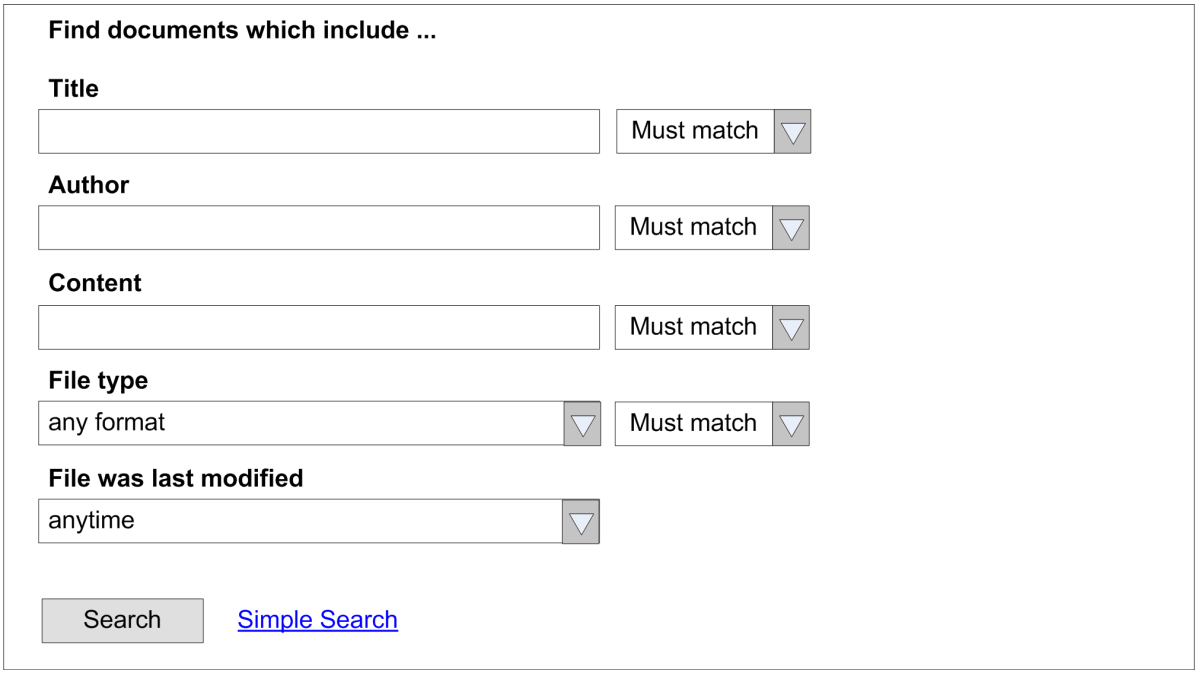

Abbildung 22: Entwurf der erweiterten Suchmaske

Der dritte Bereich besteht aus den Punkten 4 und 5 der Anforderungen. Im Punkt 4 soll der Anwender aus einer Ergebnisliste Dokumente öffnen können. Dazu werden Dokumente in der Ergebnisliste als Link präsentiert. Klickt der Anwender auf den Link, so wird das entsprechende Dokument in einem weiteren Fenster geöffnet. Bei dem fünften und letzten Punkt soll die Ergebnisliste neben den Links zu Dokumenten zusätzliche Informationen wie z.B. Autor, Titel, Dateityp und Dateigröße sowie einen Ausschnitt aus dem Inhalt des Dokuments präsentierten. Der Detailgrad der Informationen ist abhängig von den Dokumenten und wie diese im Index abgespeichert worden sind. Betrachtet man die Dokumente aus dem Dateisystem-Index, so können folgende Informationen angezeigt

werden: Content, Author, Title, Filetype, Lastmodified, Length, Name und Path. Diese Felder können mit den in Tabelle 3 vorgestellten Methoden gefüllt und im Index abgespeichert werden. Abbildung 23 zeigt den Entwurf der Webseite mit Suchergebnissen.

Abbildung 23: Entwurf der Suchergebnisanzeige

5.4 Entwurf der Suchoberfläche als Eclipse-RCP-Plugin

Die Entwurfsentscheidungen der einfachen und erweiterten Suchmaske sowie die Ergebnisdarstellung der Suchoberfläche als Webseite gelten auch für diese Suchoberfläche. Lediglich konzeptuelle Unterschiede zwischen der Realisierung als Webseite sowie RCP-Plugin werden in diesem Abschnitt erläutert. Hierzu gehören zusätzliche Anforderungen, die nur für diese Suchoberfläche relevant sind. Folgende Anforderungen werden zusätzlich beim Entwurf beachtet:

- Das System soll eigenständig, also auch ohne Zugang zum Firmen-Intranet, lauffähig sein. Ein hierdurch beschränkter Funktionsumfang ist hinnehmbar, jedoch zu dokumentieren.
- Szenarien des Gesamtsystems sollen mit der Suchmaschine des Search-Layers kommunizieren und die gefundenen Elemente darstellen können.

Laut erstem Punkt soll das Expertensystem lokal und eigenständig laufen. Dies wird durch Realisation als RCP-Anwendung erreicht. Dementsprechend wird die Suchoberfläche als RCP-Plugin implementiert, das anschließend im Gesamtsystem installiert werden kann. Zusätzlich zum Plugin wird der Index (mitsamt relevanten Dokumenten) ausgeliefert. Bei dem zweiten Punkt der Anforderungen soll das RCP-Plugin einen Service anbieten, der von anderen Komponenten des Expertensystems genutzt werden kann für das Durchsuchen des Inhalts von mit dem System ausgelieferten Dokumenten.

Ein Beispiel: ein Anwender bekommt Erläuterungen zu einem Sachverhalt in einem Fenster angezeigt. In einem zweiten Fenster werden zu dem Sachverhalt relevante Dokumente aufgelistet. Im Hintergrund hat das Expertensystem (genauer: die Erläuterungskomponente) automatisch mit Hilfe des Such-Services nach relevanten Dokumenten gesucht. Die Ergebnisse werden in einer View präsentiert. Die Form der Ergebnisdarstellung entspricht vollständig dem im vorherigen Abschnitt zuvor oben beschriebenen Entwurf. Abbildung 24 zeigt die Integration des RCP-Plugins zur Suche im Gesamtsystem.

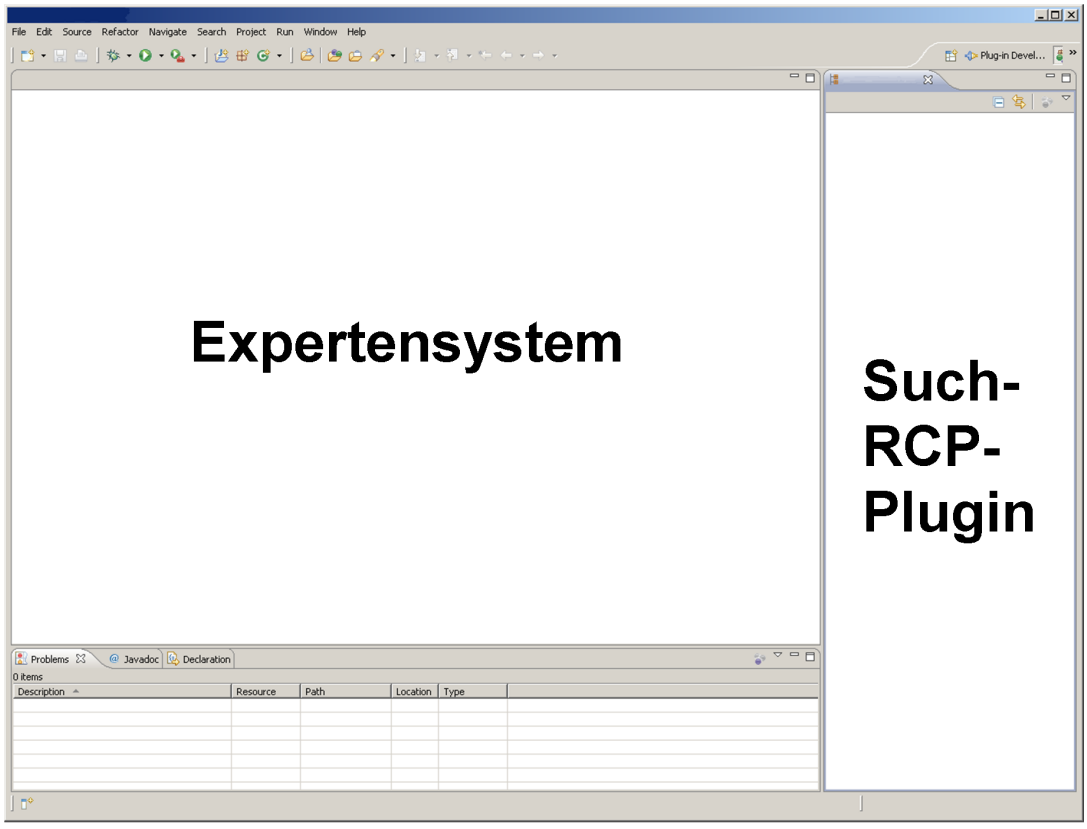

Abbildung 24: Fensteranordnung des Expertensystems und des Such-RCP-Plugins

6 Implementierung des Systems

In diesem Kapitel wird die Implementierung der zuvor entworfenen Softwarekomponenten vorgestellt. Dazu stellt eine allgemeine Übersicht die entwickelten Softwarekomponenten dar und geht kurz auf die verwendeten Technologien ein. Anschließend folgt eine detaillierte Beschreibung der Implementierung der einzelnen Komponenten.

6.1 Allgemeine Übersicht

Die entwickelte Software teilt sich, wie im Kapitel 5.1 vorgestellt, in drei Komponente auf. Die Auflistung zeigt die Reihenfolge, bei der Implementierung vorgegangen wurde:

- Dateisystem-Indexer
  - Erstellung von JUnit Test-Klassen für die unterstützten Parser-Komponenten
  - Implementierung der Parser-Klassen gegen die Test-Klassen
  - Anschließend Erstellung des Indexers
- Webseite als Benutzeroberfläche für den Anwender im Firmen-Intranet
  - Einfache Suchmaske als Servlet implementiert
  - Erweiterte Suchmaske als Servlet implementiert
- Eclipse-RCP-Plugin als integrierte Benutzeroberfläche für den Anwender des Expertensystems
  - Erstellung einer Eclipse-View mit einfacher und erweiterter Suchmaske
  - Such-OSGI-Service zum Plugin hinzugefügt

Bei der Umsetzung der Komponenten wurden im Wesentlichen die Technologien Java 6, Java Servlet, Eclipse RCP/Plugin-Platform 3.3, JUnit 4, Subversion und Subclipse, Checkstyle, Apache Lucene 2.3.2, Apache POI 3.1 sowie PDFBox 0.7.3 verwendet.

- Java 6: Für die Implementierung dieser Arbeit wurden wichtige Eigenschaften von Java 6 [JAVA] verwendet wie Generics und das neue Collections-Framework. Die Entscheidung für Java 6 als Entwicklungsplattform wurde seitens der Projektleitung für das Gesamtprojekt getroffen und dieser Arbeit zugrunde gelegt.
- Java Servlet: Die Implementation der Intranet-Weboberfläche wurde auf Grundlagen von Servlets, die ein fester Bestandteil der Java-Enterprise-Edition [JAVAEE] Anwendungsserver ist, realisiert. Die Ausführung dieser erfolgt in der Web-Container-Umgebung Apache Tomcat [TOMCA] in der Version 5.5 sowie Eclipse internen Jetty Webserver [JETTY] Version 5.1.
- Eclipse RCP/Plugin 3.3: Die Implementation des Expertensystems wurde auf Grundlage der Rich-Client-Platform von Eclipse in Version 3.3 realisiert. Diese diente gleichzeitig als Entwicklungsplattform. Ein Besonderes Augenmerk auf die Modularisierung der wesentlichen Komponenten wurde bei der Implementierung gelegt, um möglichst Unabhängig der Eclipse- Rich-Client-Platform zu sein. Abschnitt 3.2 gibt einen Überblick über die zum Einsatz kommende Rich-Client-Platform.
- JUnit 4: Bei JUnit [JUNIT] handelt es sich um ein Test-Framework, das einzelne Units (hier sind damit Java-Klassen gemeint) auf deren Funktionalität überprüft. Dieses Test-Framework hat besonders im Eclipse-Umfeld eine hohe Verbreitung, da einer der Haupt-Entwickler, Erich Gamma, ebenfalls für die Entwicklung von Eclipse verantwortlich ist. Für die Realisierung der Tests wurde JUnit in der Version 4 eingesetzt.
- Subversion und Subclipse: Für sämtliche Quelltexte wurde das Open-Source Versionsverwaltungssystem Subversion in einem gemeinsamen Projekt-Repository verwendet. Dabei kam das Eclipse-Plugin Subclipse [SUBCL] in der Version 1.4 und die Subversion Software [SUBV] in der Version 1.5 zum Einsatz.
- Checkstyle: [CHESTY] Dieses Werkzeug dient der statischen Codeanalyse von Java-Quelltext zur Überprüfung von Code-Konventionen. Mit festgelegten Regeln wird überprüft, ob für jede Klasse bzw. jede Methode Java-Dokumentation existiert, Namenskonventionen eingehalten werden, „Best Practices“ hinsichtlich des Quelltext-Aufbaus und einfache Klassendesign-Prinzipien eingehalten werden. Im Rahmen des Gesamtprojekts wurde ein einheitlicher Regelsatz definiert und verwendet. Die

verwendete Version ist Checkstyle 4.4, das sich mit dem Eclipse-Plugin „eclipse-cs“ [CHESTYE] in Eclipse integrieren lässt.

- Apache Lucene 2.3.2: Die Kernfunktionalitäten der Volltextsuche wurden mit der Apache Lucene realisiert. Abschnitt 3.1 widmet sich ausführlich dieser Java-Bibliothek.
- Apache POI 3.1: Bei POI [POI] handelt es sich um eine Java-Bibliothek, die zum Lesen und Schreiben von Dokumenten aus Microsofts Office-Paket eingesetzt wird. Die Methoden zur Text-Extraktion sowie zur Gewinnung von Metadaten kommen im Indexer zum Einsatz. Aufgrund der Qualität des Projekts und seiner Verbreitung ist Apache POI inzwischen eines der Top-Level-Projekte der Apache Foundation.
- Apache PDFBox 0.7.3: Bei PDFBox [PDFBOX] handelt es sich um eine Java-Bibliothek, die innerhalb dieser Arbeit zur Text-Extraktion sowie zur Gewinnung von Metadaten von PDF-Dokumenten verwendet wird. Die Bibliothek wird derzeit durch ein Incubator-Projekt der Apache Foundation bereitgestellt.

Der gesamte Sourcecode teilt sich in vier Packages auf und kann auf der beiliegenden CD-ROM genutzt werden. Die Namen der Packages werden im Folgenden vorgestellt.

- **de.dlr.xps.lucene.index:** dieses Package besteht aus Klassen, Test-Klassen sowie Testdateien, die zum Erstellen des Dateisystem-Indexers benötigt werden.
- **de.dlr.xps.lucene.search.servlet:** dieses Package beinhaltet Servlet-Klassen. Diese repräsentieren die Suchmasken der Webseite.
- **de.dlr.xps.search:** dieses Package beinhaltet Klassen, die die Benutzeroberfläche sowie ein OSGI-Service des Search-RCP-Plugins repräsentieren.
- **de.dlr.xps.search.servlet:** dieses Package beinhaltet Servlet-Klassen, die die jeweiligen Suchmasken repräsentieren. Diese Servlet-Klassen sind an den Eclipse internen Jetty Webserver angepasst.

6.2 Implementierung Dateisystem-Indexer

Der Dateisystem-Indexer wurde als ein Kommandozeilen-Werkzeug entwickelt. Dieses kann automatisch und im Hintergrund Dokumente extrahieren und sie anschließend zu einem Index hinzufügen. Der Indexer besteht aus mehreren Parser-Klassen. Für jedes in der Anforderung genannte Dateiformat wurden Test-Klassen, Testdateien sowie Parser-Klassen erstellt. Diese werden im Folgenden vorgestellt.

Dateiformat	**Parser-Test-Klasse**	**Parser-Klasse**
Text (.txt)	TEXTParserTest.java	TEXTParser.java
Rich Text Format (.rtf)	RTFParserTest.java	RTFParser.java
Hypertext Markup Language (.html)	HTMLParserTest.java	HTML2Parser.java
Extensible Markup Language (.xml)	XMLParserTest.java	XMLParser.java
Portable Document Format (.pdf)	PDFParserTest.java	PDF2Parser.java
Microsoft Excel (.xls)	EXCELParserTest.java	EXCELParser.java
Microsoft Word (.doc)	WORDParserTest.java	WORDParser.java
Microsoft PowerPoint (.ppt)	PPTParserTest.java	PPTParser.java
OpenDocument-Presentation (.odp)	ODPParserTest.java	OOParser.java
OpenDocument-Spreadsheet (.ods)	ODSParserTest.java	OOParser.java
OpenDocument-Text (.odt)	ODTParserTest.java	OOParser.java
XML Calc (Tabellenkalkulation) (.sxc)	SXCParserTest.java	OOParser.java
XML Impress (Präsentationen) (.sxi)	SXIParserTest.java	OOParser.java
XML Writer (Textverarbeitung) (.sxw)	SXWParserTest.java	OOParser.java

Tabelle 4: Zuordnung der Dateiformate zu den Test- sowie Parser-Klassen

Obige Tabelle 4 zeigt die Zuordnung der Dateiformate zu den einzelnen Test- sowie Parser-Klassen. Zu jedem Dateiformat wurde eine eigene Test-Klasse erstellt. Die Test-Klassen nutzen das Junit-Testframework, um die Funktionalität der entsprechenden Parser-Klassen zu gewährleisten.

Wie man in der Tabelle erkennen kann, werden die Dateiformate odp, ods, odt, sxc, sxi und sxw von nur einer Parser-Klasse verarbeitet. Bei diesen Dateiformaten handelt es sich um die Datentypen der OpenOffice Software. Diese sechs Dateiformate besitzen eine einfache offene Struktur in Form mehrere gezippte XML-Dokumente, in dem Inhalt einfach zu extrahieren ist.

Die grundsätzliche Vorgehensweise bei der Entwicklung wird exemplarisch anhand des Dateiformats Portable Document Formats (PDF) erläutert. Die Vorgehensweise bei den anderen Dateiformaten ist analog: Zunächst werden in einer Test-Klasse (im Beispiel: PDFParserTest.java) alle möglichen Ereignisse und Ausnahmen, die beim Parser einer entsprechenden Datei beim Indexieren auftreten können, getestet. Die PDFParserTest-Klasse besteht aus elf Methoden, die mit Hilfe von fünf PDF-Testdateien getestet werden. Als Erstes werden die PDF-Testdateien und anschließend die Testmethoden der PDFParserTest-Klasse beschrieben. Tabelle 5 zeigt die Eigenschaften der für die Tests erstellten PDF-Testdateien.

Name der PDF Testdatei	**Beschreibung**
TestPDF.pdf	Diese PDF-Datei besteht aus einer Seite. Der Inhalt dieser Seite lautet „Test“. Zusätzlich sind noch Metainformationen wie Autor „Test Author“ und Titel „Test Title“ abgespeichert.
TestPDFCorrupt.pdf	Diese PDF-Datei kann nicht gelesen werden.
TestPDFEmpty.pdf	Diese PDF-Datei kann zwar gelesen werden, aber der Inhalt, der Titel sowie das Autorfeld sind leer.
TestPDFEncrypted.pdf	Diese PDF-Datei ist passwortgeschützt.
TestPDFMorePages.pdf	Diese PDF-Datei besteht aus zwei Seiten. Auf der ersten Seite steht „Test Page1“ und auf der zweiten Seite „Test Page2“.

Tabelle 5: Beschreibung der PDF-Testdateien

6.2.1 Implementierung der PDFParserTest-Klasse

Die fünf PDF-Testdateien repräsentieren mögliche Ausnahmen sowie normale Eigenschaften einer PDF-Datei, die beim Parsen auftreten können. Die konkreten Tests werden in Methoden gekapselt. Diese werden in Hinblick auf bestimmte Ausnahmen oder Eigenschaften erstellt. So entstehen bei der PDFParserTest-Klasse elf Testmethoden, die im Folgenden in Tabelle 6 beschrieben werden.

Name der Testmethode	Beschreibung
setUp()	Diese Methode dient zur Vorbereitung auf den Testlauf. Hier werden die fünf PDF-Testdateien zusammen mit den PDF2Parser geladen.
testFileExists()	Dieser Test überprüft, ob die fünf PDF-Testdateien überhaupt existieren.
testFileCanBeRead()	Dieser Test überprüft, ob die Dateien gelesen werden können. Die Leseberechtigung wird überprüft.
testFileCanBeParsed()	Hier kommt die Parser-Klasse erst zum Einsatz. Es wird überprüft, ob der Inhalt der fünf PDF-Dateien geparst werden kann.
testFileEmpty()	Diese Methode geht auf die Datei „TestPDFEmpty.pdf“ ein. Hier wird überprüft, ob der PDF2Parser eine leere Datei richtig interpretiert und leeren Inhalt zurückliefert.
testFileContentTest()	Diese Methode nutzt den PDF2Parser mit der Datei „TestPDF.pdf“. Hier wird überprüft, ob der Parser den erwarteten Inhalt „Test“ ausgibt.
testFileContentMorePages()	Diese Methode nutzt den PDF2Parser mit der Datei „TestPDFMorePages.pdf“. Hier wird überprüft, ob der Parser mehrere Seiten richtig extrahieren kann.
testFileEncrypted()	Diese Methode nutzt den PDF2Parser mit der Datei „TestPDFEncrypted.pdf“. Dabei soll der PDF2Parser erkennen, dass es sich dabei um eine passwortgeschützte Datei handelt. Anschließend soll der Parser ordnungsgemäß den Vorgang abbrechen.
testFileCourrupt()	Der PDFParser nutzt hier die Datei „TestPDFCorrupt.pdf“. Der Parser soll auch hierbei erkennen, dass eine Datei nicht gelesen werden kann und ordnungsgemäß den Vorgang abbrechen.
testGetTitle()	Diese Methode testet, ob der Parser, der die „TestPDF.pdf“ Datei geladen hat, den Titel „Test Title“ ausgibt.
testGetAuthor()	Diese Methode testet, ob der Parser, der die Datei „TestPDF.pdf“ geladen hat, den Auhor „Test Author“ ausgibt.

Tabelle 6: Testmethoden der PDFParserTest-Klasse

Es folgt eine detaillierte Beschreibung der PDFParserTest-Klasse anhand eines Quelltextausschnitts. Mit der Methode setUp() werden alle nötigen Klassen erstellt, die zum Testen notwendig sind. Hier werden die Parser erzeugt. Beim Erstellen von Parsern muss die zu extrahierende Datei als Parameter an diesen übergeben werden. Deshalb wird zu jeder PDF-Testdatei ein Objekt zu jedem zu testenden Parser erzeugt. Der relevante Programmierausschnitt sieht wie folgt aus:

```
...
public class PDFParserTest extends TestCase {
...
protected void setUp() throws Exception {
      super.setUp();
      IndexerLogger.getLogger().setLevel(Level.OFF);
      this.file = new File("testfiles/pdf/testPDF.pdf");
      this.fileEmpty = new File("testfiles/pdf/testPDFEmpty.pdf");
      this.fileMorePages = new File("testfiles/pdf/testPDFMorePages.pdf");
      this.fileEncrypted = new File("testfiles/pdf/testPDFEncrypted.pdf");
      this.fileCorrupt = new File("testfiles/pdf/testPDFCorrupt.pdf");
      this.parser = new PDF2Parser(file);
      this.parserEmpty = new PDF2Parser(fileEmpty);
      this.parserMorePages = new PDF2Parser(fileMorePages);
      this.parserEncrypted = new PDF2Parser(fileEncrypted);
      this.parserCorrupt = new PDF2Parser(fileCorrupt);
}
```

Mit der Methode testFileExisits() wird für die fünf angegebenen Testdateien überprüft, ob sie als File-Objekte im Dateisystem existieren. Ein File-Objekt repräsentiert eine Datei oder ein Verzeichnis. Mittels der File-Methde exists() wird die Existenz einer Datei überprüft. Die Testmethode assertTrue() erwartet als Rückgabewert den booleschen Wert „true“. Der zugehörige Programmcode sieht wie folgt aus:

```
...
public void testFileExists() {
    assertTrue(file.exists());
    assertTrue(fileEmpty.exists());
    assertTrue(fileMorePages.exists());
    assertTrue(fileEncrypted.exists());
    assertTrue(fileCorrupt.exists());
}
```

Mit der Methode testFileCanBeRead() werden die fünf Testdateien überprüft, ob sie als File-Objekt gelesen werden können. Bei diesem Test wird die File-Methode canRead() eingesetzt. Der zugehörige Programmcode sieht wie folgt aus:

```
...
public void testFileCanBeRead() {
    assertTrue(file.canRead());
    assertTrue(fileEmpty.canRead());
    assertTrue(fileMorePages.canRead());
    assertTrue(fileEncrypted.canRead());
    assertTrue(fileCorrupt.canRead());
}
```

Mit der Methode testFileCanBeParsed() werden die fünf in der Methode setUp() erstellten Parser getestet. Der Test verläuft erfolgreich, wenn die Parser „parser“, „paserEmpty“ und „parserMorePages“ erfolgreich die entsprechenden Dateien geparst haben und die Parser „parserEncrypted“ sowie „parserCorrupt“ nicht erfolgreich waren. In den beiden letzten Fällen soll getestet werden, ob die Methode parserDocument() der PDF2Parser-Klasse passwortgeschützte und korrupte Dateien erkennen kann und diese nicht weiter bearbeitet. Der Programmcode dieser Methode sieht folgendermaßen aus:

```
...
public void testFileCanBeParsed() {
    assertTrue(parser.parseDocument());
    assertTrue(parserEmpty.parseDocument());
    assertTrue(parserMorePages.parseDocument());
    assertFalse(parserEncrypted.parseDocument());
    assertFalse(parserCorrupt.parseDocument());
}
```

Mit der Methode testFileEmpty() wird der Parser „parserEmpty“ überprüft, ob dieser eine leere PDF-Datei parsen kann und der Inhalt der Parser Methode getContent() exakt einen leeren String zurückliefert. Bei der assertEquals-Testmethode werden zwei Werte miteinander verglichen. Der Test ist erfolgreich, wenn beide Werte übereinstimmen. Siehe Programmcode:

```
...
public void testFileEmpty() {
    parserEmpty.parseDocument();
    assertEquals("", parserEmpty.getContent());
}
```

In der Methode testFileContentTest() kommt der Parser „parser“ zum Einsatz. Dabei liefert die Methode getContent() der Parser-Klasse den Inhalt der testPDF.pdf Datei. Hat der Parser richtig gearbeitet, so liefert die Methode genau den String „Test“ aus. Der Test ist demnach erfolgreich gelaufen, weil der erwartete Wert mit der Ausgabe übereinstimmt. Der zugehörige Programmcode lautet:

```
...
public void testFileContentTest() {
    parser.parseDocument();
    assertEquals("Test", parser.getContent());
}
```

In der Methode testFileContentMorePages() wird der Parser „parserMorePages“ eingesetzt. Dieser soll den Inhalt „Test Page1“ und „Test Page2“ liefern. Siehe Programmcode:

```
...
public void testFileContentMorePages() {
    parserMorePages.parseDocument();
    assertEquals("Test Page1\r\nTest Page2", parserMorePages.getContent());
}
```

Mit der Methode testFileEncypted() soll der Parser „parserEncrypted“ bei einer passwortgeschützten Datei den Wert „false“ zurückliefern. Hier wird die Methode parseDocument() getestet, bei der geschütze Dateien wunschgemäß ignoriert werden. Es soll verhindert werden, dass ein Parser bei einer geschützen Datei nicht den gesamten Indexierungsprozess aufhält. Siehe entsprechenden Programmcode:

```
...
public void testFileEncrypted() {
    assertFalse(parserEncrypted.parseDocument());
}
```

Mit der Methode testFileCorrupt() wird die Parser Methode parseDocument() auf korrupte Dateien getestet. Dabei soll der Wert „false“ zurückgeliefern werden, falls eine Datei korrupt ist. Siehe Programmcode:

```
...
public void testFileCorrupt() {
    assertFalse(parserCorrupt.parseDocument());
}
```

Mit der Methode testgetTitle() soll die Parser Methode getTitle() den erwarteten Wert „Test Title“ aus der Testdatei „testPDF.pdf“ zurückliefern. Siehe Programmcode:

```
...
public void testgetTitle() {
    parser.parseDocument();
    assertEquals("Test Title", parser.getTitle());
}
```

Mit der Methode testgetAuthor() soll die Parser Methode getAuthor() den erwarteten Wert „Test Author“ aus der Testdatei „testPDF.pdf“ zurückliefern. Siehe Programmcode:

```
...
public void testgetAuthor() {
    parser.parseDocument();
    assertEquals("Test Author", parser.getAuthor());
}
```

Auch zu jedem weiteren Dateiformat wurden Test-Klassen entwickelt, die die gleichen Testmethoden beinhalten, wie die gerade vorgestellten. Damit nicht jede Test-Klasse einzeln ausgeführt werden muss, wurde eine Test-Suite-Klasse „AllParserTests.java“ erstellt. Abbildung 25 zeigt die JUnit-Testergebnisse aller Parser Test-Klassen.

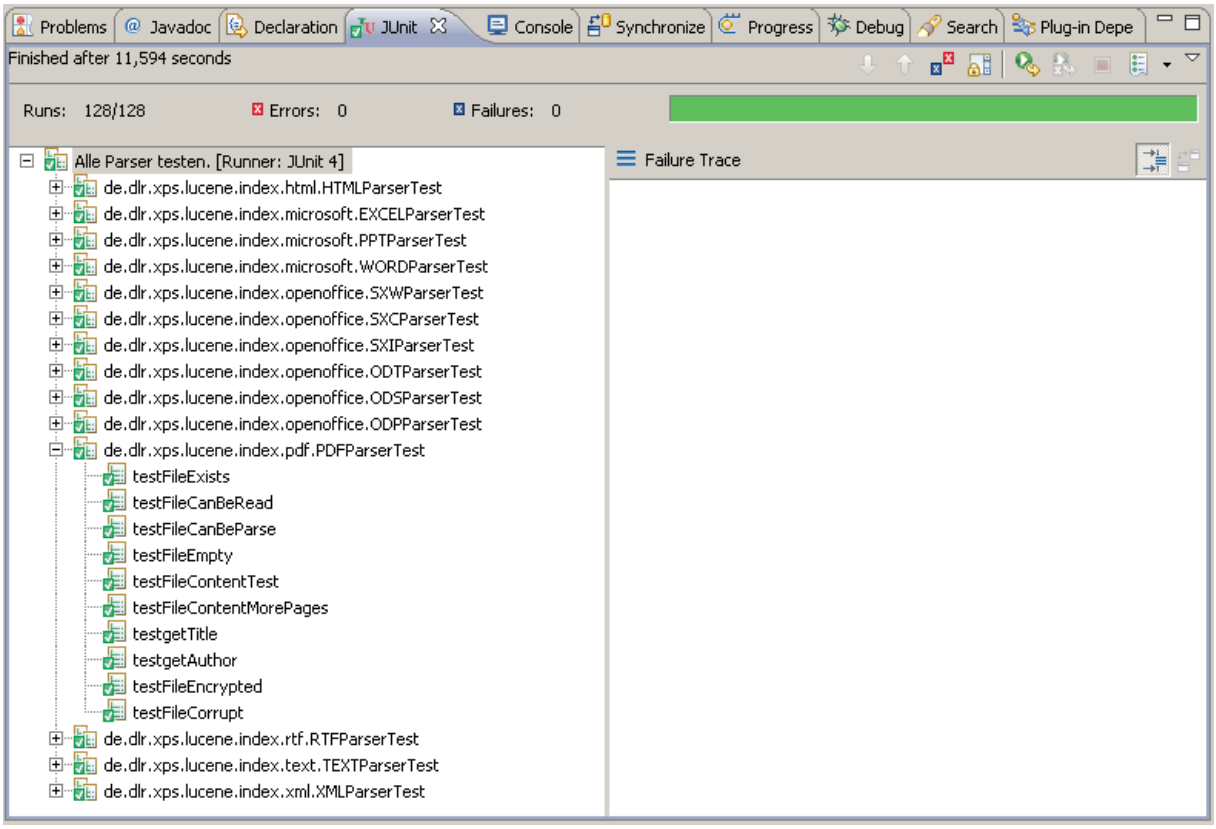

Abbildung 25: JUnit-Testergebnisse aller Parser Test-Klassen

6.2.2 Implementierung der PDF-Parser-Klasse

Im Folgenden wird der in der PDFParserTest-Klasse verwendete PDF Parser vorgestellt. Die Beschreibung dieser Klasse steht exemplarisch für die anderen Parser-Klassen. Es werden Methoden, Abhängigkeiten, Funktionsweisen sowie die verwendeten APIs der Klasse anhand eines Programmcodes beschrieben. Im Kapitel Entwurf zeigt Abbildung 19 das UML-Klassendiagramm der Parser-Klassen, die implementiert worden sind. Alle Parser-Klassen erben Methoden der abstrakten Klasse „Parser“. Diese Klasse bietet Methoden an, die in jedem Parser gleich funktionieren und Methoden die als abstrakt gekennzeichnet sind. Die abstrakten Methoden müssen von konkreten Parser-Klassen überschrieben werden. Abbildung 26 zeigt die abstrakte Parser-Klasse „Parser“ zusammen mit der PDF-Parser-Klasse „PDF2Parser“.

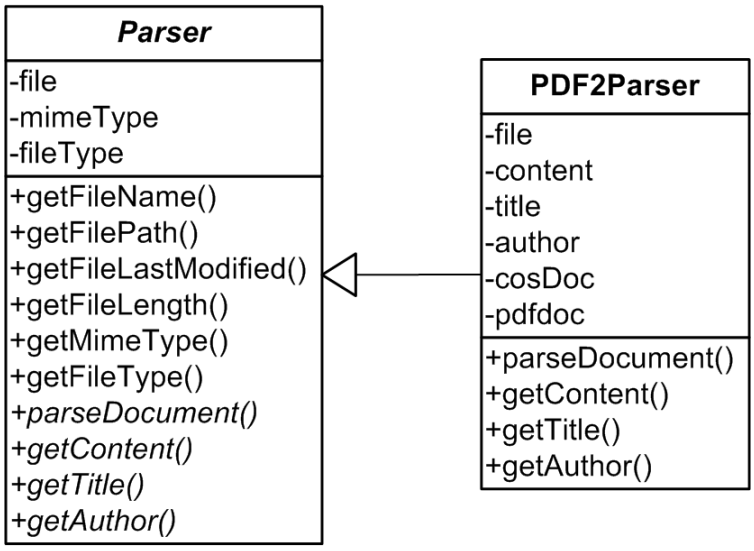

Abbildung 26: Klassendiagramm der abstrakten Parser- und PDF2Parser-Klassen

Als Erstes wird die abstrakte Parser-Klasse und anschließend die PDF2Parser-Klasse erklärt. Die abstrakte Klasse besitzt einen Konstruktor, dem eine File-Klasse als Parameter übergeben wird. Im Konstruktor wird der File-Typ sowie der MIME-Type der File-Klasse erkannt und den beiden Variablen fileType und mimeType zugewiesen, schließlich wird das übergebene Objekt der Variablen file zugewiesen. Im Folgenden werden in der Tabelle die nicht-abstrakten Methoden der Parser-Klasse vorgestellt.

Name der Methode	Beschreibung
getFileName()	Diese Methode liefert einen String zurück, der den Namen der Datei repräsentiert. Hierbei wird die Java File-Methode getName() benutzt.
getFilePath()	Diese Methode liefert den Pfad der Datei als String zurück. Die Java File-Methode getPath() wird hier eingesetzt.
getFileLastModified()	Diese Methode liefert mit Hilfe der Java File-Methode lastModified() einen long Wert, der das Datum der letzten Modifikation der Datei repräsentiert.
getFileLength()	Mit Hilfe der Java File-Methode length() wird die Größe der Datei als long Wert zurückgeliefert.
getMimeType()	Diese Methode liefert den Wert der Variablen mimeType zurück.
getFileType()	Diese Methode liefert den Wert der Variablen fileType zurück.

Tabelle 7: Methoden der abstrakten Parser-Klasse, die nicht abstrakt sind

Die in Tabelle 7 beschriebenen Methoden stehen per Vererbungsmechanismen jedem konkreten Parser zur Verfügung. Die Methoden parseDocument(), getContent(), getTitle() sowie getAuthor() sind als abstrakte Methoden gekennzeichnet und müssen von einem konkreten Parser überschrieben werden. Im Folgenden wird dies am Beispiel der PDF2Parser-Klasse beschrieben. Der PDF-Parser erbt von der abstrakten Parser-Klasse. Über einen entsprechend parametrisierten Konstruktor, wird eine Datei als Parameter übergeben. Im Konstruktor selbst werden nur Variablen wie file, content, title sowie author gesetzt. Der zugehörige Programmcode sieht wie folgt aus:

```
...
public class PDF2Parser extends Parser {
    private String content;
    private String title;
    private String author;
    private COSDocument cosDoc;
    private PDDocument pdfdoc;
    private File file;

    public PDF2Parser(final File file) {
        super(file);
```

```
        this.file = file;
        this.content = new String("");
        this.title = new String("");
        this.author = new String("");
    }
```

Mit der Methode parseDocument() wird die übergebene Datei „file“ überprüft, ob diese gelesen werden kann. Anschließend wird sie in der privaten Methode parseDocument(InputStream) ausgeführt. Die private Methode liefert ein COSDocument zurück, das ein PDF-Dokument im Arbeitsspeicher repräsentiert. Das COSDocument wird mit dem PDFParser, der Java-Bibliothek PDFBox erstellt. Die Datei wird dem PDFBox Parser übergeben und dieser liefert mit der Methode getDocument() das COSDocument zurück. Jetzt kann mit der Methode isEncrypted() abgefragt werden, ob das Dokument passwortgeschützt ist. Die Methode parseDocument() liefert das Ergebnis „false“, wenn die PDF-Datei nicht gelesen werden kann oder passwortgeschützt ist. Des Weiteren kommt ein PDFTextStripper zum Einsatz, der mit der Methode getText() den Inhalt der PDF-Datei extrahiert und diesen in der Variable content ablegt. Die Variablen author und title werden mit einer PDDocumentInformation Klasse mit den Methoden getAuthor und getTitle gesetzt. Anschließend wird mit der privaten Methode closeCOSDocument das COSDocument geschlossen. Ist alles ordnungsgemäß gelaufen, so liefert die Methode parseDocument() der PDF2Parser-Klasse „true“ zurück. Das bedeutet, dass die PDF-Datei extrahiert worden ist und keine Fehler dabei entstanden sind. Hier folgt der Programmcode dieser Methode:

```
...
public final boolean parseDocument() {
   try {
    if (file.canRead()) {
       cosDoc = parseDocument(new FileInputStream(file));
       if (!cosDoc.isEncrypted()) {
          PDFTextStripper txtstripper = new PDFTextStripper();
          pdfdoc = new PDDocument(cosDoc);
          if ( txtstripper.getText(pdfdoc) != null )
               this.content = txtstripper.getText(pdfdoc);
          PDDocumentInformation pdfinfo = pdfdoc.getDocumentInformation();
          if( pdfinfo.getAuthor() != null )
              this.author = pdfinfo.getAuthor();
          if( pdfinfo.getTitle() != null )
```

```
                this.title = pdfinfo.getTitle();
            closeCOSDocument(cosDoc);
            return true;
        } else {
            closeCOSDocument(cosDoc);
            IndexerLogger.getLogger().error(file + " encrypted");
            return false;
        }
    } else {
        IndexerLogger.getLogger().error(file + " can not read");
        return false;
    }
...
```

Mit der Methode parseDocument() wurden alle benötigten Inhalte sowie Informationen extrahiert. Dazu wurden die Variablen „content“, „title“ sowie „author“ mit den entsprechenden Werten belegt. Die Aufgabe der Methoden getContent(), getTitle() sowie getAuthor() ist es, die jeweiligen Variablen als String zurückzuliefern. Der entsprechender Programmcode sieht wie folgt aus:

```
...
public final String getContent() {
    return this.content.trim();
}
public String getAuthor() {
    return this.author.trim();
}
public String getTitle() {
    return this.title.trim();
}
```

Der PDF-Parser nutzt die PDFBox 0.7.3 API [PDFBOX], um Inhalt, Titel sowie Autor aus einer PDF-Datei als Text zu extrahieren. Der Einsatz solcher API's verringert den Implementierungsaufwand. Im Folgenden wird in der Tabelle eine Übersicht dargestellt, die alle implementierten Parser mit der jeweils zugehörigen API beschreibt.

Parser-Klasse	Die zugehörige API
TEXTParser	In dieser Klasse werden mit Hilfe der BufferedReader-Klasse der Java API [JAVA] die Textdateien zeilenweise gelesen.
RTFParser	Hier kommt die RTFEditorKit-Klasse der Java API zum Einsatz. Mit Hilfe der Methoden getProperty() sowie getText() dieser Klasse, werden Informationen abgeholt.
HTML2Parser	In dieser Klasse wird die HTMLParser-Klasse der Apache Lucene API verwendet. Diese besitzt Methoden wie getTitle() und getMetaTags(), die den Zugriff zu den benötigten Informationen erleichtern.
XMLParser	In dieser Klasse wird der XML Parser der Java API verwendet. Dabei werden die Werte aller Knoten mit dem Standard-Parser herausgezogen.
PDF2Parser	In dieser Klasse wird die Apache PDFBox 0.7.3 API [PDFBOX] verwendet. Insbesondere kommt dabei der PDFTextStripper zum Einsatz, der das Extrahieren der benötigten Inhalte einfach macht.
EXCELParser	In dieser Klasse wird die Apache POI 3.1 API [POI] verwendet. Es können Excel-Version von 97 bis 2007 geöffnet werden. In dem EXCELPaser kommt insbesondere der ExcelExtractor zum Einsatz. Dieser erlaubt es, mit Methoden wie getSummaryInformation() sowie getText() auf die benötigten Informationen zuzugreifen.
WORDParser	Diese Klasse nutzt wie Excel die Apache POI 3.1 API. Es können Word-Dateien in den Versionen von 97 bis 2007 geöffnet werden. Hier wird insbesondere der WordExtractor verwendet, der die gleichen Methoden verwendet wie der ExcelExtractor.
PPTParser	Diese Klasse nutzt auch die Apache POI 3.1 API. Damit können PowerPoint-Dateien in den Versionen 97 bis 2007 gelesen werden. Der PowerPointExtractor verhält sich wie der Word- oder ExcelExtractor.
OOParser	In dieser Klasse werden ausschließlich Methoden der Java API verwendet. Die implementierten OpenOffice-Dateien besitzen die gleiche komprimierte Struktur. Die Datei muss erst einmal entpackt werden. Anschließend kann man mit dem XML-Parser der Java API auf die Dateien meta.xml sowie content.xml zugreifen. Mit diesem Parser werden Dateien im OpenDocument-Format gelesen.

Tabelle 8: Übersicht aller implementierten Parser und der jeweils dazugehörigen API

6.2.3 Implementierung des Indexers

Im Folgenden wird die Funktionsweise des Indexers, die in Abbildung 20 als Programmablaufplan dargestellt wurde, mit dem zugehörigen Programmcode erläutert. Nach dem alle benötigten Parser implementiert worden sind, können die Dokumente in einem Index indexiert werden. Die main() Methode der Klasse Main erzeugt den LuceneIndexer. Dabei werden die Eingabeparameter an diese weitergegeben. In der LuceneIndexer-Klasse werden mit Hilfe der JSAP Java Simple Argument Parser API die Parameter bearbeitet. Die erwarteten Parameter sind: Indexverzeichnis (wo der Index abgelegt wird) und ein oder mehrere Dokument-Verzeichnisse, die indexiert werden sollen. Zusätzlich kann ein Flag „-f“ gesetzt werden. Bei dieser Option wird der Index überschrieben, falls er in dem angegebenen Indexverzeichnis existiert. Anschließend werden die Dokument-Verzeichnisse in einer privaten Methode bearbeitet. Dabei wird eine Hash-Tabelle mit Dateiort und dem Modifikationsdatum erstellt. Hier soll sichergestellt werden, dass nur Dateien in der Tabelle eingetragen sind. Diese Methode sieht wie folgt aus:

```
...
private static void readFileStructur(String[] filearg) {
   for (int x = 0; x < filearg.length; x++) {
       File dir = new File(filearg[x]);
       // check user rights
       if (dir.canRead()) {
          if (dir.isDirectory()) {
             String[] rec = dir.list();
             for (int i = 0; i < rec.length; i++) {
               rec[i] = dir.getAbsolutePath() +
      System.getProperty("file.separator") + rec[i];
             }
             readFileStructur(rec);
          } else {
             HASHTABLE_FILES_DATA.put(dir.toString(),
      Long.toString(dir.lastModified()));
          }
        } else {
          IndexerLogger.getLogger().error(dir + " can not read");
        }
   }
}
```

Als Nächstes wird überprüft, ob ein Index im angegebenen Indexverzeichnis vorhanden ist. Tritt der Fall ein, dass der Index bereits vorhanden ist, so wird der Flag „FLAG_CREATE_INDEX“ auf „false“ gesetzt. Anschließend wird mit Hilfe der IndexReader-Klase der vorhandene Indexer ausgelesen. Dazu der Programmcode:

```
...
IndexReader indexReader = IndexReader.open(PATH_INDEX);
int anzahlDocs = indexReader.numDocs();
for (int y = 0; y < anzahlDocs; y++) {
    if (!indexReader.document(y).getFields().isEmpty()) {
      HASHTABLE_FILES_INDEX.put(indexReader.document(y).
      getField("path").stringValue(), indexReader.document(y).
      getField("lastmodified").stringValue());
    }
}
indexReader.close();
```

Anschließend wird die Index-Hash-Tabelle mit der Datei Hash-Tabelle verglichen. Dabei entstehen drei Tabellen mit Dateien, die im Index aktualisiert, neu hinzugefügt und aus dem Index gelöscht werden sollen. Falls kein Index vorhanden ist oder der Flag auf „-f“ gesetzt worden ist, so wird der Flag „FLAG_CREATE_INDEX“ auf „true“ gesetzt. Anschließend kann der Index erstellt werden. Dazu wird die Klasse IndexWriter benutzt.

```
...
INDEXWRITER = new IndexWriter(PATH_INDEX, ANALZYER, FLAG_CREATE_INDEX);
Date startIndexingTime = new Date();
IndexerLogger.getLogger().info("Start Indexing at " +
      startIndexingTime.toString());
FILESYSTEM_FACTORY = new FileIndexerFactory();
```

Die IndexWriter Klasse erwartet drei Parameter: das Indexverzeichnis, den zu verwendeten Analyzer und ob der Index beim Erzeugen dieser erstellt werden soll. Hier wird der StandardAnalyzer zur Analyse der Dokumente verwendet. Zusätzlich wird die Factory-Klasse FileIndexerFactory() erstellt. Diese dient dazu konkrete Parser zu den entsprechenden Dateien zu erstellen. Im weiteren Verlauf des Indexers werden die erstellten Hash-Tabellen je nach Zustand der Variabel „FLAG_CREATE_INDEX“ abgearbeitet. Bei einem neuen Index wird nur eine Hash-Tabelle in der privaten Methode addIndex() abgearbeitet. Bei dieser Hash-Tabelle handelt es sich um Datei die in den Index hinzugefügt werden sollen. Bei einem

vorhanden Index werden Hash-Tabellen, mit zu löschenden, zu aktualisierenden sowie mit neuen Dateien bearbeitet. Im Folgenden werden die drei zuständigen privaten Methoden addIndex(), updateIndex() sowie delIndex() erläutert. Die Methode addIndex() fügt die Dokumente der übergebenen Hash-Tabelle mit Hilfe der Faktory-Klasse „FileSystemFactory“ dem Index zu. Der Programmcode sieht wie folgt aus:

```
...
private static void addIndex(Collection<String> filecoll) {
    try {
      Iterator<String> fileCollIter = filecoll.iterator();
      while (fileCollIter.hasNext()) {
         String filePath = (String) fileCollIter.next();
         File file = new File(filePath);
         Document doc = FILESYSTEM_FACTORY.getDocument(file);
         if (doc.getFields().size() != 0) {
             IndexerLogger.getLogger().info("Add to Index " + filePath);
             INDEXWRITER.addDocument(doc);
             INDEXWRITER.flush();
         }
         FLAG_CREATE_INDEX = false;
    }
```

Die Methode updateIndex() überprüft erst einmal die Werte der übergebenen Hash-Tabelle, ob sich Dateien tatsächlich verändert haben. Dabei wird das Datum einer Datei als Vergleichskriterium benutzt. Nur bei einer Veränderung wird die entsprechende Datei im Index aktualisiert. Dazu wird die Methode updateDocument() der IndexerWriter Klasse benutzt. Siehe Programmcode:

```
...
private static void updateIndex(Collection<String> filecoll) {
    try {
        Iterator<String> fileCollIter = filecoll.iterator();
        while (fileCollIter.hasNext()) {
          String filePath = (String) fileCollIter.next();
          File file = new File(filePath);
          long dataFileLastMod =
      Long.valueOf(HASHTABLE_FILES_DATA.get(filePath)).longValue();
          long indexFileLastMod =
      Long.valueOf(HASHTABLE_FILES_INDEX.get(filePath)).longValue();
```

```
          if (file.exists()) {
           if (dataFileLastMod != indexFileLastMod) {
             Document doc = FILESYSTEM_FACTORY.getDocument(file);
             IndexerLogger.getLogger().info("Update of Index " + filePath);
             INDEXWRITER.updateDocument(new Term("path", filePath), doc);
             INDEXWRITER.flush();
           }
          }
        }
```

Die Methode delIndex() löscht, die in der übergebenen Hash-Tabelle vorhanden Dateien aus dem Index heraus. Der Programmcode hierfür sieht wie folgt aus:

```
...
private static void delIndex(Collection<String> filecoll) {
    try {
        Iterator<String> fileCollIter = filecoll.iterator();
        while (fileCollIter.hasNext()) {
           String filePath = (String) fileCollIter.next();
           IndexerLogger.getLogger().info("Delete from Index " + filePath);
           INDEXWRITER.deleteDocuments(new Term("path", filePath));
           INDEXWRITER.flush();
        }
```

Beim Hinzufügen oder Aktualisieren der Dokumente wird die Factory-Methode getDocument() benutzt. Diese Methode ist in der abstrakten Klasse IndexerFactory erstellt worden. Diese Methode erstellt ein Lucene Dokument mit den entsprechenden Feldern die im Index abgespeichert werden sollen. Anschließend liefert diese Methode das erstellte Lucene Dokument zurück.

```
...
public Document getDocument(File file) {
    Document ldoc = new Document();
    Parser fdoc = createParser(file);
    if (fdoc != null) {
     if (fdoc.parseDocument()) {
      if (fdoc.getContent() != null) {
       ldoc.add(new Field("filetype", fdoc.getFileType(),
       Field.Store.YES,       Field.Index.UN_TOKENIZED));
       ldoc.add(new Field("mimetype", fdoc.getMimeType(),
```

```
        Field.Store.YES,       Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("name", fdoc.getFileName(), Field.Store.YES,
        Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("path", fdoc.getFilePath(), Field.Store.YES,
        Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("lastmodified",
        Long.toString(fdoc.getFileLastModified()),
        Field.Store.YES, Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("length",
        Long.toString(fdoc.getFileLength()),     Field.Store.YES,
        Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("title", fdoc.getTitle(), Field.Store.YES,
        Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("author", fdoc.getAuthor(), Field.Store.YES,
        Field.Index.UN_TOKENIZED));
        ldoc.add(new Field("content", fdoc.getContent(),
        Field.Store.YES,       Field.Index.TOKENIZED));
             }
         }
    }
    return ldoc;
}
```

Alle Felder werden im Index abgespeichert und bis auf „content“ nicht in Tokens zerteilt. Die Namen der Felder sind selbsterklärend. Im Feld „content“ wird der Originaltext im Index abgespeichert, aber zugleich auch für die Volltextsuche in Tokens zerteilt. Die Methode getDocument() verwendet die abstrakte Methode createParser(). Hierbei wird die Aufgabe des Extrahieren an die konkrete Faktory-Klasse übergeben. Die FileIndexerFactory ist eine konkrete Factory-Klasse der IndexerFactory. In dieser wird die Methode createParser() überschrieben. Diese hat die Aufgabe einen entsprechenden Parser zu einer Datei auszuwählen. Anschließend liefert diese den Parser zurück. Der Ausschnitt dieser Methode sieht wie folgt aus:

```
...
public Parser createParser(File file) {
    Parser doc = null;
    String filenamelow = file.getName().toLowerCase();
    if (filenamelow.endsWith(".pdf")) {
        doc = new PDF2Parser(file);
```

```
    } else if (filenamelow.endsWith(".txt")) {
        doc = new TEXTParser(file);
    } else if (filenamelow.endsWith(".doc")) {
        doc = new WORDParser(file);
    } else if (filenamelow.endsWith(".sxw") || filenamelow.endsWith(".sxi")
    ...
    return doc;
}
```

Während der Indexierung werden Dateiname sowie die Ereignisse der Prozesse in der Konsole protokolliert. Nachdem die Indexierung für alle Dokumente abgeschlossen ist, wird ein Resultat des gesamten Prozesses angezeigt. Dabei wird die Dauer der Indexierung, Dauer der Optimierung sowie die Anzahl der Dokumente im Index angezeigt.

6.2.4 Nutzung des Indexers

Mit dem Aufruf „java -jar Indexer.jar“ bzw. des Skripts startIndexer.bat oder startIndexer.sh kann der Indexer gestartet werden. Das Programm erwartet folgende Parameter:

```
[-f|--force] : Overwrite the Index.
(-i|--indexpath) <indexpath> : Path to index.
datapath1 datapath2 ... datapathN : Path to data you would like to index.
```

Der Parameter „indexpath“ repräsentiert das zu erstellende Indexverzeichnis und muss angegeben werden. Der Parameter „datapath“ repräsentiert ein oder mehrere Verzeichnisse, die indiziert werden sollen. Es können beliebig viele Verzeichnisse, aber mindestens eins muss angegeben sein. Der Parameter „-f“ erstellt den zu erstellenden Index neu. Dieser Parameter kann optional gesetzt werden. Damit der Indexer starten kann, muss ein Unterverzeichnis „lib“ mit den entsprechenden API's existieren. Dieses muss sich in dem gleichen Verzeichnis befinden, in dem der Indexer „Indexer.jar“ sich befindet. Abbildung 27 zeigt einen kompletten Durchlauf des Indexers. Dabei erzeugt der Indexer im angegebenen Verzeichnis „d:\demo\index\“ einen Index für Dokumente aus dem Verzeichnis „d:\demo\daten\“ und dessen Unterverzeichnisse.

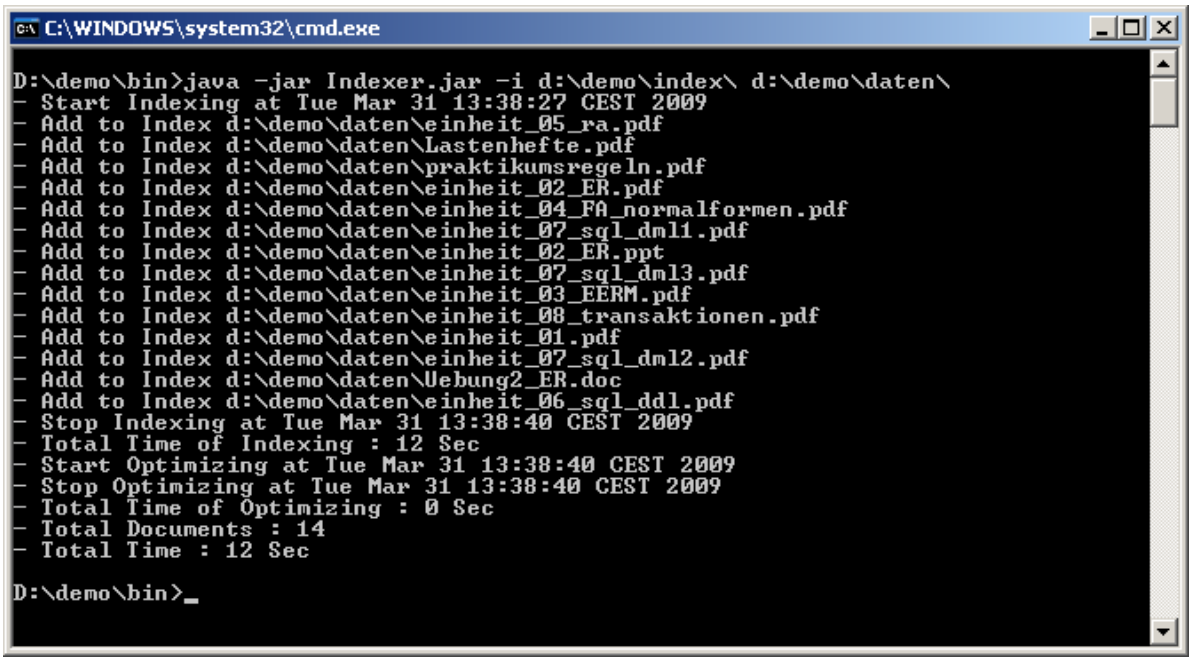

Abbildung 27: Indexierung von Demo Dokumenten mit dem Indexer

6.3 Implementierung der Suchmasken als Webseite

Im Folgenden werden die beiden im Entwurf skizzierten Suchmasken vorgestellt. Diese wurden auf Basis der Java Servlet Technologie implementiert. Jede Suchmaske besteht aus einer Servlet-Klasse, die in diesem Kapitel mit Programmcode erläutert werden. Die Ergebnisse werden aus den jeweiligen Servlet-Klassen erstellt. Abbildung 28 zeigt die implementierte einfache Suchmaske.

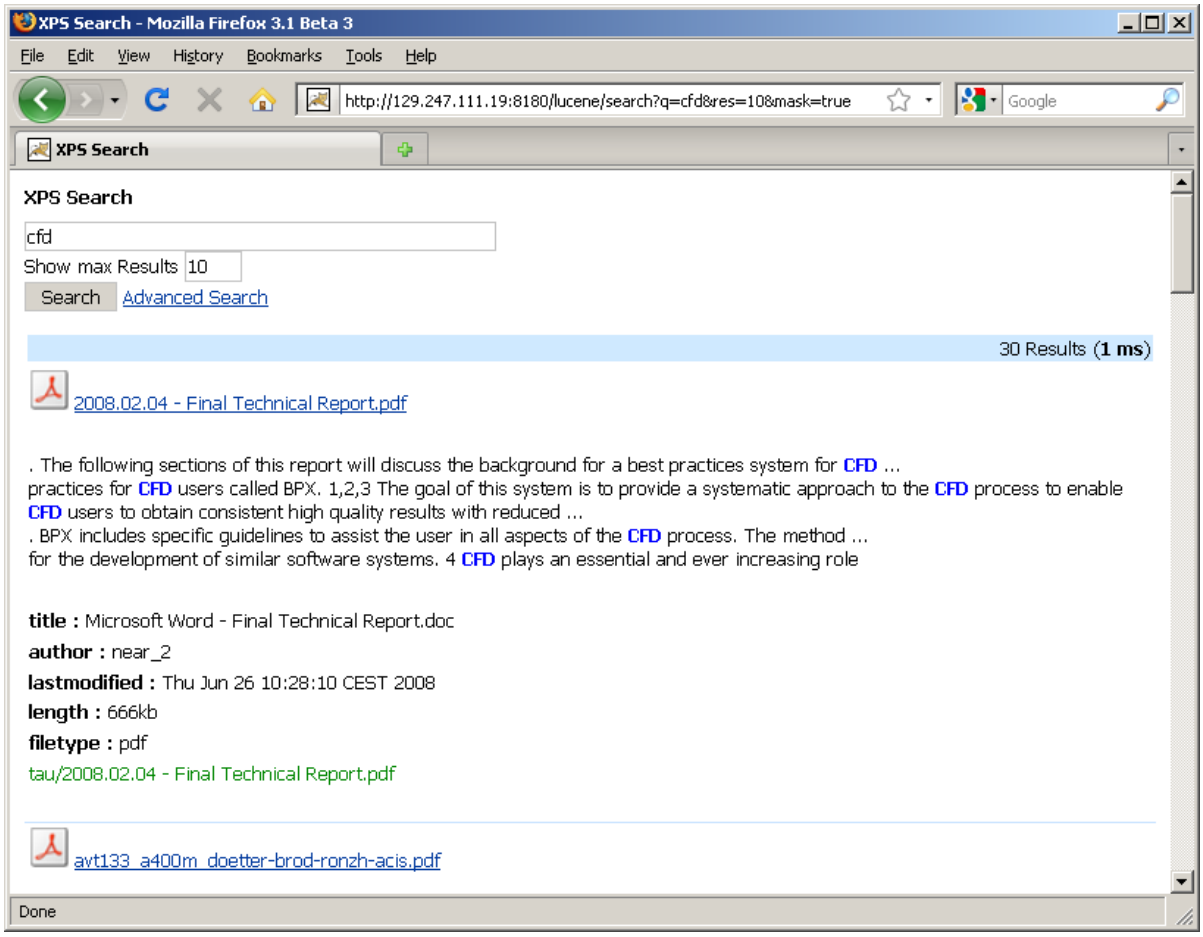

Abbildung 28: Browseransicht der einfachen Suchmaske mit Suchergebnissen

Die Suchmaske besteht im Wesentlichen aus einem Formular mit zwei Eingabefeldern, einem Button und einem Link. Das erste Eingabefeld dient zur Eingabe der Suchterme. Hier können beispielsweise Wörter mit Booleschen Operatoren miteinander verknüpft werden. Das zweite Eingabefeld erfordert die Eingabe einer Zahl. Diese Zahl begrenzt die Ausgabe der gefundenen Resultate. Tippt man die Zahl 0 ein, so werden alle gefunden Ergebnisse angezeigt. Mit dem Button startet man den Suchprozess mit der angegebenen Anfrage. Der Anwender kann die Suchmaske wechseln, in dem er den Link auf die erweiterte Suchmaske klickt. In Abbildung 28 werden auch die Ergebnisse einer Suchanfrage angezeigt. Die Anzeige der Resultate wurden, wie im Entwurf skizziert realisiert. Dabei werden Informationen zur einer Suchanfrage, wie folgt dargestellt. Ein direkter Link zum Dokument wird angezeigt. Der Anwender kann durch Klicken auf den Link das entsprechende Dokument öffnen. Des Weiteren wird ein Ausschnitt aus dem Inhalt des Dokuments angezeigt und die dort gefundenen Suchterme mit farblich in blau hervorgehoben. Zusätzlich werden

Informationen wie Titel, Autor, letzte Änderung, die Größe, Dateityp sowie der Pfad des jeweiligen Dokuments angezeigt. Die Anzahl der gefundenen Ergebnisse, sowie die Dauer des Suchprozesses werden unter der Suchmaske angezeigt.

Im Folgenden werden Codeausschnitte aus der Servlet-Klasse gezeigt, um die Funktionsweise der einfachen Suchmaske zu erläutern. Beim Initialisieren der Servlet-Klasse in einem „Servlet-Container“ wie beispielsweise Apache Tomcat 5.5 wird die Methode init() ausgeführt. Dabei werden Parameter aus der XML Datei web.xml gelesen. Der wesentliche Ausschnitt dieser Datei sieht wie folgt aus:

```
...
<servlet-name>search</servlet-name>
<servlet-class>SearchServlet</servlet-class>
<init-param>
    <param-name>index_directory</param-name>
    <param-value>/local-home/xps/tauindexer/index</param-value>
</init-param>
```

Der Parameter „index_directory“ beinhaltet einen String. Dabei handelt es sich um ein Verzeichnis, dass auf den Speicherort des Index zeigt. Die Angabe dieses Parameters ist für die Funktion des Servlets notwendig. Die Datei web.xml findet man bei einer Apache Tomcat Installation im Unterverzeichnis WEB-INF des jeweiligen Projektes. Das Formular aus der Suchmaske wird mit Klicken auf den Button „Search“ mit der GET Methode an sich selber abgeschickt. Die Felder der Suchmaske werden mit der doGet() Methode abgefragt. Die Suchanfrage steht im dem Parameter „q“ und die maximale Anzahl der anzuzeigenden Resultate im Parameter „res“. Nach Überprüfen dieser wird mit Hilfe der Lucene Klassen IndexSearcher und MultiFieldQueryParser, der Suchprozess vorbereitet:

```
...
IndexSearcher is = new IndexSearcher(DIR_INDEX);
QueryParser qp = new MultiFieldQueryParser(new String[] {"content",
"name"}, ANALYZER );
try {
  query = qp.parse(query_string);
  HIGHLIGHTER = new Highlighter(new SimpleHTMLFormatter("<strong
      style=\"color:blue\">","</strong>"), new QueryScorer(query));
```

Ein IndexSearcher wird mit dem Parameter des Indexverzeichnisses erstellt. Zusätzlich wird ein QueryParser erstellt, der über die Felder „content“ und „name“ automatisch bei einer Anfrage durchsuchen wird. Felder können bei einer Anfrage explizit angegeben werden, z.B. durch eine Anfrage wie „filetype:pdf +content:cfd“. Diese findet PDF Dokumente, die das Wort „cfd“ beinhalten. Bei der Analyse der Suchanfrage wird der gleiche Analyzer benutzt, wie beim Indexer. Dabei handelt es sich um den StandardAnalyzer von Lucene.
Die vom Anwender formulierte Suchanfrage muss mit Hilfe des QueryParsers in ein system-spezifische Query umgewandelt werden. Bei einer einfachen formulierten Anfrage wie „cfd“, wandelt das implementierte System die Query ins „content:cfd name:cfd“ um. Hierbei wird das Wort „cfd“ in den Feldern „content“ und „name“ gesucht. Beinhaltet eins von den Feldern dieses Wort, so wird das jeweilige Indexdokument als Treffer angezeigt. Mit Hilfe der Highlighter-Klasse werden die Suchterme im Inhaltsausschnitt eines gefundenen Dokuments formatiert. Dabei werden die Suchterme durch schriftsetzerische (fett Schrift) und farbliche Hervorhebung (blau) kenntlich gemacht. Anschließend wird die Suche mit der erstellten Query angestoßen. Dies erfolgt mit dem Aufruft der Methode search().

```
Hits hits = is.search(query);
```

Der Suchprozess liefert eine Klasse Hits zurück. Hierbei handelt es sich um die Resultate der Suchanfrage. Die Anzahl der gefundenen Dokumente wird mit hits.length() abgefragt und in der Variablen „results“ abgespeichert. Anschließend werden die Hits iterativ abgearbeitet. Dazu der Programmcode:

```
...
for (int i = 0; i < results; i++) {
      Document doc = hits.doc(i);
      writer.print(hitOutput(doc, query));
}
```

Jedes einzelne Dokument wird zusammen mit der Anfrage in die private Methode hitOutput() übergeben und ausgeführt. Diese Methode liefert die in der Abbildung 28 präsentierten Ergebnisse für jedes einzelne Dokument. Die Ausgabe der Servlet-Klasse kann mit der Cascading-Style-Sheets Datei layout.css formatiert werden. Diese befindet sich im Unterverzeichnis „css“ des Web-Projekts. Im Folgenden wird die erweiterte Suchmaske vorgestellt. Abbildung 29 zeigt diese in einer Browseransicht.

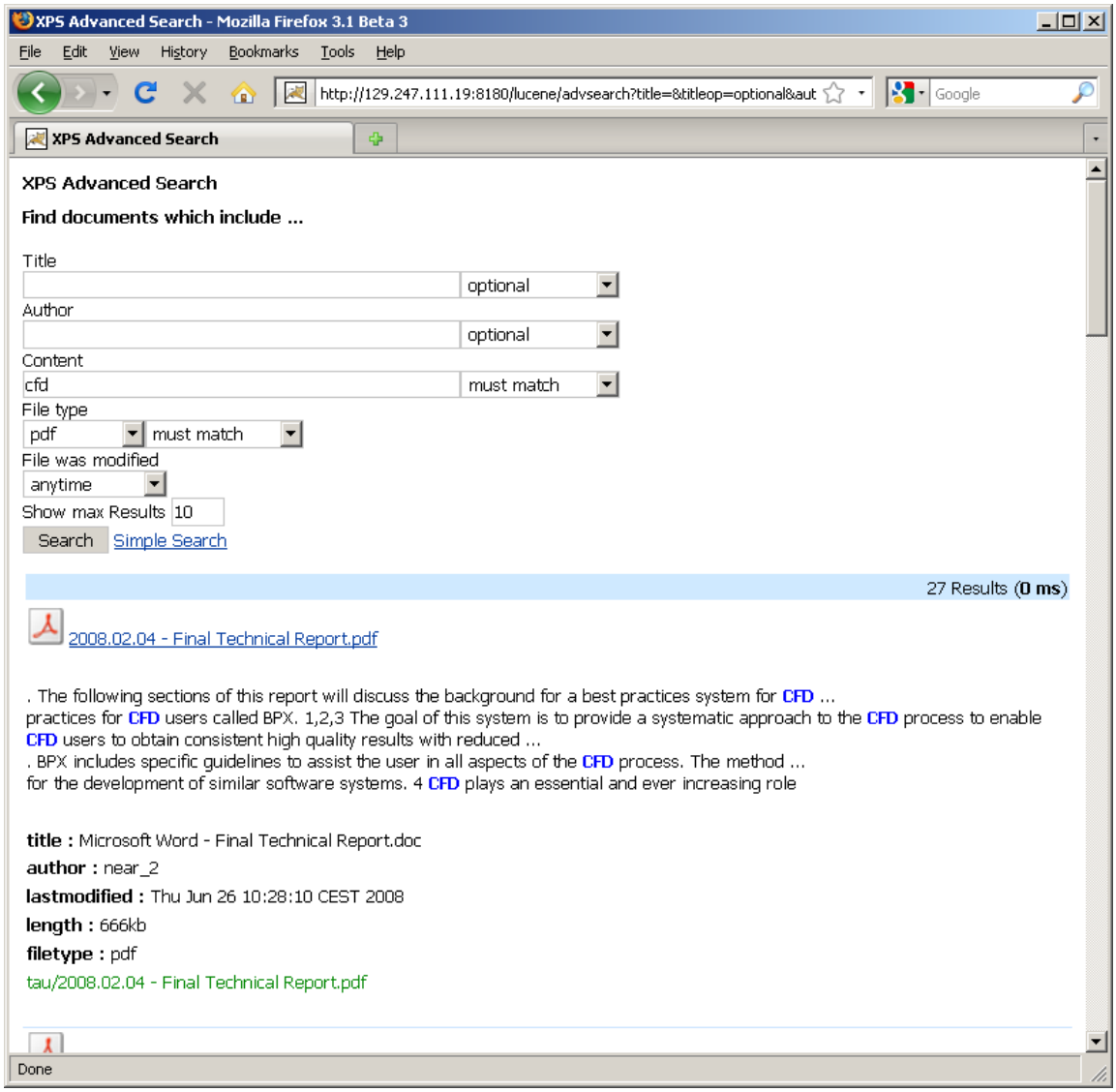

Abbildung 29: Browseransicht der erweiterten Suchmaske mit Suchergebnissen

Die erweiterte Suchmaske unterscheidet sich gegenüber der einfachen Suchmaske in der Form der Eingabefelder und der verwendeten QueryParser. Die Ergebnisanzeige ist identisch mit der Anzeige, der einfachen Suchmaske. Lediglich die wesentlichen Unterschiede der Suchmasken werden im Folgenden vorgestellt. Diese Suchmaske orientiert sich an den gängigen Suchmasken im Internet. Zweck dieser Suchmaske ist, gezielt in vordefinierten Feldern zu suchen. Dabei kann der Anwender mit Operatoren bestimmen, ob ein gesuchtes Wort in einem Feld gefunden werden muss „must match“, das Wort „optional“ im Feld gefunden wird oder aber nicht vorkommen darf „must not match“. Diese Operatoren können für die Felder „Title“, „Author“, „Content“ und „File type“ gewählt werden. Am Beispiel des „Title“ Feldes wird die Implementierung der Operatoren vorgestellt:

```
...
BooleanQuery boolQuery = new BooleanQuery();
if (hashmapRequest.get("title") != null) {
   if (!hashmapRequest.get("title").equals("")) {
     Query titleQuery = new TermQuery(new
      Term("title",hashmapRequest.get("title")));
        if (hashmapRequest.get("titleop").equals("must match")) {
            boolQuery.add(titleQuery,Occur.MUST);
        }
        if (hashmapRequest.get("titleop").equals("must not match")) {
            boolQuery.add(titleQuery,Occur.MUST_NOT);
        }
        if (hashmapRequest.get("titleop").equals("optional")) {
            boolQuery.add(titleQuery,Occur.SHOULD);
        }
    }
}
```

Bei dieser Suchmaske wird die BooleanQuery verwendet. Zu dieser können mehrere einzelne Anfragen mit Operatoren „MUST“, „MUST_NOT“ und „SHOULD“ hinzugefügt werden. Daraus entsteht eine Kette aller Anfragen, die vom System in eine Query umgewandelt wird. Diese wird anschließend vom IndexSearcher ausgeführt. In der Auswahlliste „File type“ können alle implementierten Dateitypen für eine Eingrenzung der Suchanfrage gewählt werden. Mit den Einträgen in der Auswahlliste „File was modified“ kann die Suchanfrage auf bestimmte Zeiträume der zu suchenden Dokumente eingegrenzt werden. Die Optionen lautet: „anytime“, „past 24 hours“, „past week“, „past month“ und „past year“. Diese Zeiträume werden von dem Zeitpunkt aus berechnet, an dem die Anfrage gestartet worden ist. Im Folgenden wird die Implementierung dieser zeitlichen-Eingrenzung am Beispiel von „past year“ gezeigt:

```
...
Calendar calNow = Calendar.getInstance();
...
if (dateValue.equals("past year")) {
    Calendar pastYear = Calendar.getInstance();
    pastYear.add(Calendar.YEAR, -1);
    Query dateQuery = new RangeQuery(new
      Term("lastmodified",Long.toString(pastYear.getTimeInMillis())),new
      Term("lastmodified",Long.toString(calNow.getTimeInMillis())),true);
```

```
    boolQuery.add(dateQuery,Occur.MUST);
}
```

Als Erstes wird das aktuelle Datum in die Variable „calNow“ gespeichert. Im nächsten Schritt wird eine weiter Variable „pastYear“ erstellt, die aber manipuliert wird. Dabei wird das Datum um ein Jahr zurückgestellt. Anschließend wird eine RangeQuery erstellt, die eine Suche im angegebenen Feld auf ein Bereich „von“ - „bis“ einschränkt. Im Beispiel wird im Feld „lastmodified“ der Suchbereich von dem Datum der Variablen „pastYear“ bis zum Datum calNow beschränkt. Der „true“ Parameter schließt die angegebenen Daten mit ein. Die RangeQuery wird in Gesamtanfrage hinzugefügt. Abschließend wird die Gesamtanfrage „boolQuery“ ausgeführt und die Ergebnisse der zusammengesetzten Anfragen werden im Browser angezeigt.

6.4 Implementierung der Suchmasken als RCP-Plugin

Im Folgenden wird die Implementierung des im Entwurf skizzierten RCP-Plugins vorgestellt. Wie bereits im Kapitel 5.4 Entwurf erwähnt, werden die Suchmasken der Webseite im Plugin verwendet. Die Implementierung dieser wurde im Kapitel 6.3 Implementierung detailliert erläutert. Die Servlet-Klassen der jeweiligen Suchmasken werden in einem lokalen „Servlet-Container“ ausgeführt. Hier kommt der Eclipse-interne Webserver Jetty zum Einsatz. Dieser wird beim Aufrufen des Plugins automatisch gestartet. Dabei werden die Servlet-Klassen und Mapping auf Dokumente im Webserver registriert. Das Mapping ist für das Öffnen der Dokumente in der Ergebnisanzeige notwendig. Das RCP-Plugin besteht aus einer View, die einen internen Browser umfasst. Mit Hilfe dessen werden die Suchmasken und die Ergebnisanzeige dem Anwender angezeigt. Der Anwender nutzt den Browser in der View in üblicher Weise. Zusätzlich der bereits beschriebenen Funktionalität stellt das RCP-Plugin einem OSGI-Service bereit, der im Folgenden erklärt wird. Abbildung 30 zeigt die Komponentenübersicht dieses Plugins.

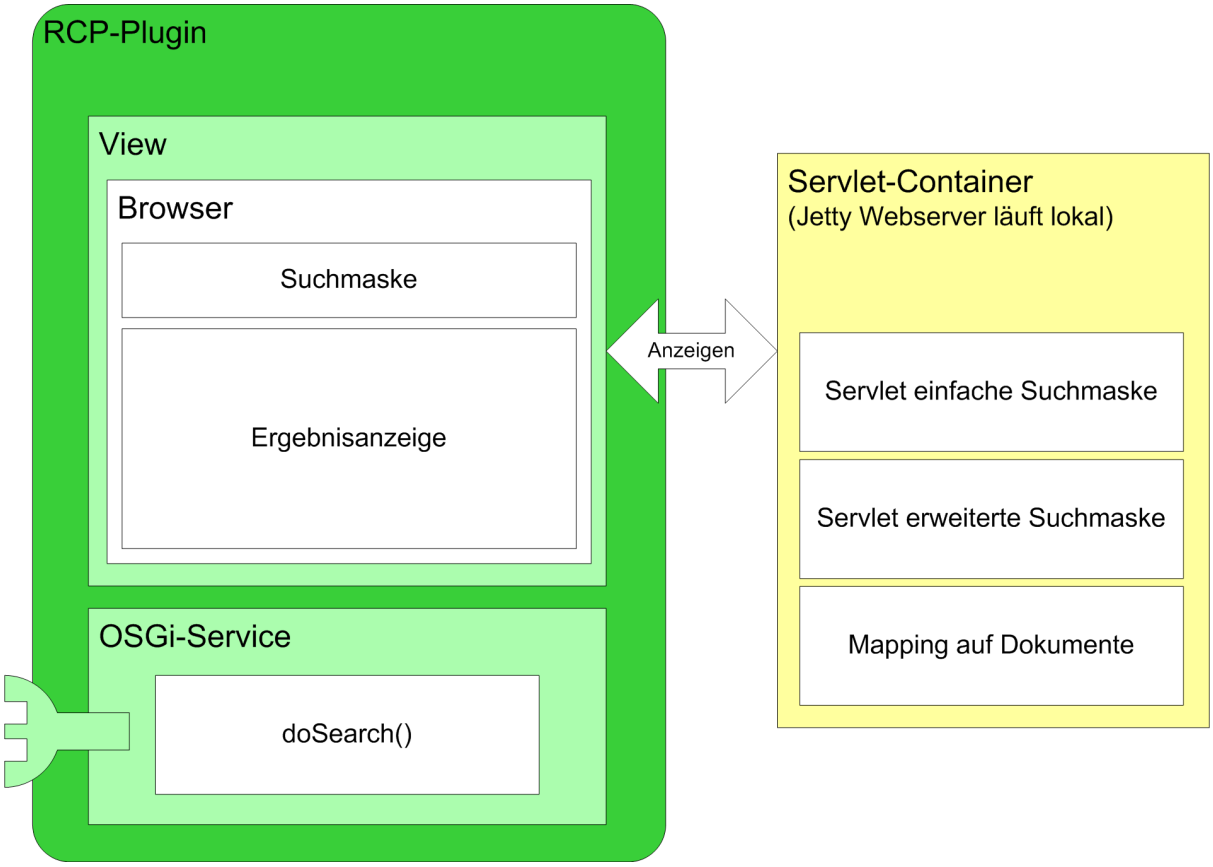

Abbildung 30: Komponentenübersicht des RCP-Plugins

Beim Starten vom RCP-Plugin wird die Methode „doSearch()“ als OSGi-Service am Gesamtsystem registriert. Diese kann von anderen Plugins des Systems aufgerufen werden. Damit andere Plugins dies tun können wurde eine Schnittstelle implementiert, die wie folgt aussieht:

```
...
public interface ISearchService {
      void doSearch(String[] searchStatements);
}
```

Die Methode dieser Schnittstelle erwartet als Eingabeparameter einen Stringarray. Es handelt sich hierbei um einen Array mit Begriffen, nach denen gesucht werden soll. In der View wurde die „doSearch()“ Methode implementiert. Dabei werden die einzelnen Begriffe in eine Anfrage zusammengefasst und an den Browser geschickt. Dieser sendet die Anfrage an den Jetty Webserver. Abschließend werden die Ergebnisse der Anfrage im internen Browser angezeigt. Abbildung zeigt die Nutzung des Such RCP-Plugins im Expertensystem. Das

Beispiel zeigt ein Szenario zum Thema „Mesh Generation Centaur“. Rechts werden mit Hilfe der doSearch() Methode kontextsensitive Suchergebnisse angezeigt.

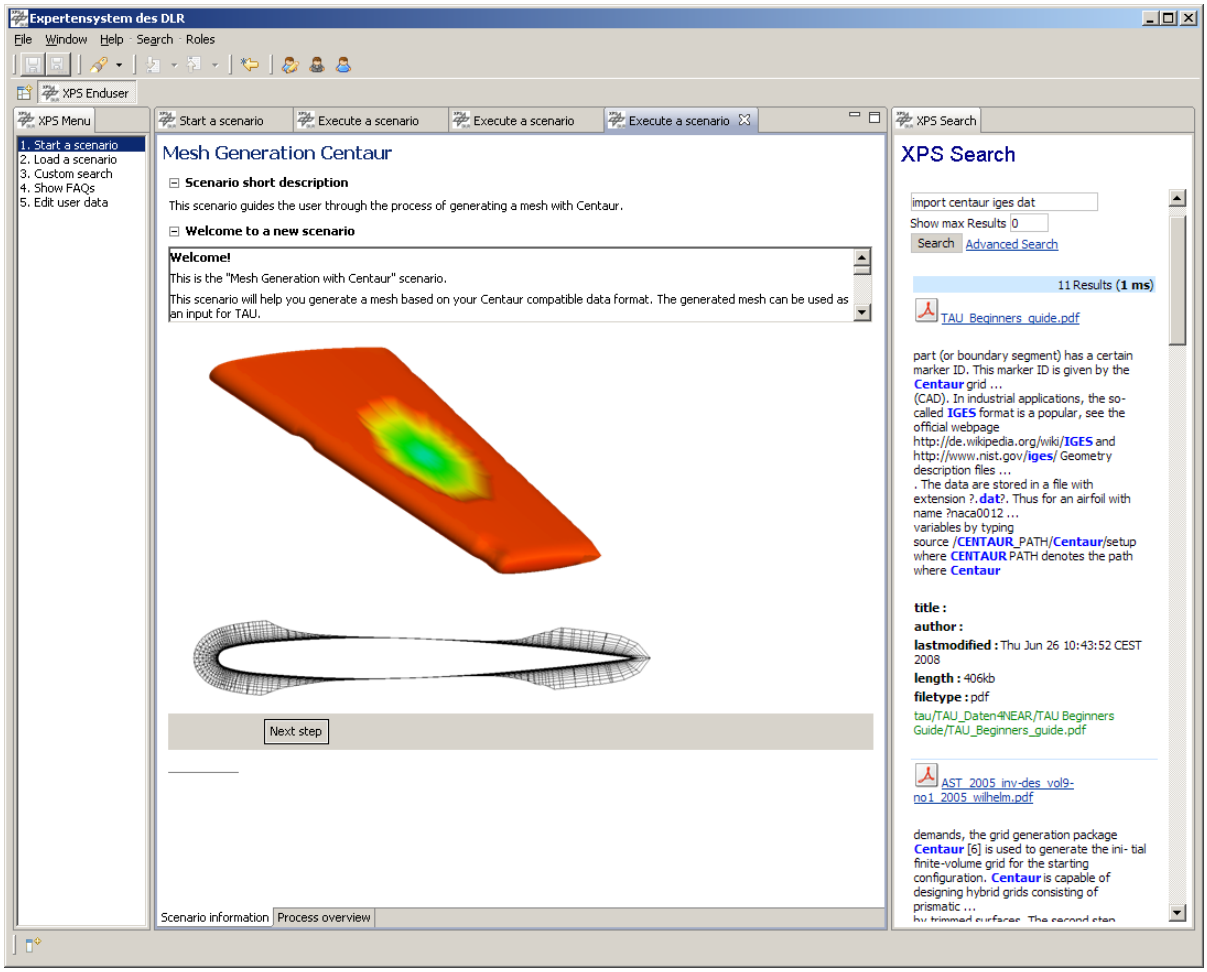

Abbildung 31: Integration des Such RCP-Plugins im Expertensystem

7 Zusammenfassung und Ausblick

Die stetig steigende Zahl wissenschaftlicher Publikationen, Bücher sowie Berichten führt zu einer Informationsflut, die kaum von einem Wissenschaftler bewältigt werden kann. Dieser kann unmöglich die gesamte Literatur seines Fachgebietes verfolgen. Deshalb ist es notwendig, die gesamte Literatur inhaltlich zu erschließen und dem Wissenschaftler die jeweils für sein aktuelles Problem relevanten Informationen zu übergeben. Diese Problematik zeigt die Relevanz von Information Retrieval-Systemen. Diese sind in der Lage, die Informationsbedürfnisse eines Anwenders gezielt zu stillen.

Die im Rahmen dieser Diplomarbeit konzipierte und implementierte Volltextsuchmaschine für wissenschaftliche Dokumente ist in der Lage, dem Endanwender die unstrukturierten Dokumente zur Verfügung zu stellen. Der Endanwender erhält damit ein Werkzeug, das ihm schnell und effizient bei der Suchen nach Informationen hilft. Die Ergebnisse einer Suchanfrage sind anwenderfreundlich gestaltet, da hier ein Teilausschnitt aus dem Inhalt des jeweiligen Dokuments dem Benutzer bei seiner Entscheidung hilft, ob ein Dokument für seine Zwecke relevant ist. Außerdem kann er durch Nutzen der erweiterten Suchmaske seine Anfragen genauer spezifizieren. Die Benutzerschnittstelle der Volltextsuche wurde als ein Plugin entwickelt, das in das parallel entwickelten Expertensystem integriert wird. Zusätzlich wurde eine Intranet-Weboberfläche entwickelt, die den Gebrauch der Volltextsuchmaschine ohne den Zugang zum Expertensystem gewährleistet.

Die Komponenten der Suchmaschine stützen sich zum großen Teil auf Programmierschnittstellen, die von der Apache Software Foundation unterstützt werden. Dadurch ist die Qualität der Programmierschnittstellen gewährleistet. Die Einhaltung von Standards bei der Entwicklung sowie der modulare Aufbau der entwickelten Software bietet ein einfach zu erweiterndes Suchsystem. Dieses kann um neue Parser erweitert werden, die neue Dokumenttypen unterstützen können. Der Einsatz der Software ist in verschiedenen Instituten des Deutschen Zentrums für Luft- und Raumfahrt geplant.

Darüber hinaus existieren mehrere Möglichkeiten, die bereits implementierte Volltextsuchmaschine zukünftig zu erweitern. So kann der Index der Suchmaschine durch

Anbindung des Indexers an die DLR-Literatur-Datenbank elib [ELIB] mit zusätzlichen Dokumenten vergrößert werden. Hierbei würde dem Endanwender bei der Suche nach Informationen eine zusätzliche Quelle zur Verfügung stehen. Des Weiteren kann die Volltextsuchmaschine um mehrere Sprachen erweitert werden. Dies kann durch Spracherkennung der jeweiligen Dokumente erfolgen. Die implementierte Volltextsuchmaschine kann zur Zeit eine vorgegebene Sprache, in der Voreinstellung Englisch, unterstützen. Zusätzlich kann im Bereich der Luft- und Raumfahrt eine entsprechende Ontologie für die automatische Analyse von Dokumenten zum Einsatz kommen. Dies würde die Effektivität der Suchmaschine erhöhen, weil eine bestimmte Begriffswelt bei der Analyse sowie bei der Suche beachtet wäre.

Literaturverzeichnis

[CHESTY] Checkstyle, http://checkstyle.sourceforge.net/, Abruf: 4.04.2009

[CHESTYE] The Checkstyle Plugin for Eclipse, http://eclipse-cs.sourceforge.net/, Abruf: 4.04.2009

[ECLI] Eclipse Foundation, http://www.eclipse.org/, Abruf: 4.04.2009

[ELIB] elib - DLR electronic library - Suche nach Publikationen innerhalb des DLR-Publikationsservers elib, http://elib.dlr.de/,

[ENT06] Entwurfsmuster von Kopf bis Fuß,Eric Freeman, Elisabeth Freeman, O'Reilly Verlag, 2006

[EQUI] Equinox, http://www.eclipse.org/equinox/, Abruf: 4.04.2009

[FUH06] Information RetrievalSkriptum zur Vorlesung im SS 06,Norbert Fuhr, , 2006

[FUH96] Fachgruppe Information Retrieval der Gesellschaft für Informatik,Norbert Fuhr, , 1996

[GER87] Information retrieval - Grundlegendes für Informationswissenschaftler,Gerard Salton ; Michael J. McGill. [Übers.: Wolfgang von Keitz]., McGraw-Hill, 1987

[HEN08] Information Retrieval 1 - Grundlagen, Modelle und Anwendungen,Andreas Henrich, , 2008

[HEN08] Information Retrieval 1 - Grundlagen, Modelle und Anwendungen, http://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/medieninformatik/Dateien/Publikationen/2008/henrich-ir1-1.2.pdf,

[JAVA] API Specifications, http://java.sun.com/reference/api/index.html, Abruf: 4.04.2009

[JAVAEE] Java Platform, Enterprise Edition, http://java.sun.com/javaee/, Abruf: 4.04.2009

[JETTY] Jetty WebServer, http://www.mortbay.org/jetty/,

[JUNIT] JUnit, http://www.junit.org/, Abruf: 4.04.2009

[KUH90] Pragmatische Aspekte beim Entwurf und Betrieb von Informationssystemen,Rainer Kuhlen, Universitätsverlag Konstanz, 1990

[LUC09] Apache Lucene, http://lucene.apache.org/java/docs/index.html,

[OSGI] OSGi Alliance, http://www.osgi.org/Main/HomePage, Abruf: 4.04.2009

[OSGISP] Die OSGi Service Platform - Eine Einführung mit Eclipse Equinox,Gerd Wütherich, Nils Hartmann, Bernd Kolb, Matthias Lübken, dpunkt.verlag, 2008

[PDFBOX] Apache PDFBox - Java PDF Library, http://incubator.apache.org/pdfbox/, Abruf: 4.04.2009

[POI] Apache POI - Java API To Access Microsoft Format Files, http://poi.apache.org/, Abruf: 4.04.2009

[REG03] Information Retrieval : Suchmodelle und Data-Mining-Verfahren für Textsammlungen und das Web,Reginald, Ferber, dpunkt-Verl., 2003

[RIC99] Modern information retrieval,Ricardo, Baeza-Yates; Berthier, Ribeiro-Neto, ACM Press, 1999

[SAL83] Introduction to modern information retreival,Salton, G.;McGill, M.J., McGraw-Hill, 1983

[SUBCL] Subclipse, http://subclipse.tigris.org/, Abruf: 4.04.2009

[SUBV] Subversion, http://subversion.tigris.org/,

[TOMCA] Apache Tomcat, http://tomcat.apache.org/, Abruf: 4.04.2009

[WOL07] Information retrieval : Informationen suchen und finden ,Wolfgang G. Stock, Oldenbourg, 2007

Anhang A

Inhalt der CD-ROM

Dieser Diplomarbeit wurde eine CD-ROM beigelegt, die folgende Verzeichnisse beinhaltet:

Quellcode

Im Quellcodeverzeichnis befindet sich der Programmcode des Dateisystem-Indexers, der Intranet Weboberfläche und des Such-RCP-Plugins.

Software

In diesem Verzeichnis befindet sich die notwendige Software, die zum Ausführen der entwickelten Programme benötigt wird. Hierbei handelt es sich um Java 6, Apache Tomcat 5.5 und Eclipse RCP-Plugin 3.3.

Programme

In diesem Verzeichnis befinden sich die aus dem Quellcode erstellten Programme und die jeweiligen Java-Bibliotheken.

Diplomarbeit

In diesem Verzeichnis befindet sich die Diplomarbeit im PDF-Format.

Erklärung über die selbständige Abfassung der Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, den 8. April 2009

Peter Pakula