Technische Universität Hamburg-Harburg
Arbeitsbereich Softwaresysteme

Projektarbeit

Prototypische Realisierung der Einbindung einer Volltextsuchmaschine in Konzeptorientierte Inhaltsverwaltungssysteme

Cord Schäfer

Betreuer:
Prof. Dr. Joachim W. Schmidt,
Dr. Hans Werner-Sehring

September 2006

Gesetzt am 29. September 2006 um 12:03 Uhr.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde weder einer anderen öffentlichen oder privaten Institution vorgelegt noch veröffentlicht.

`Cord Schäfer`
Hamburg, den 24. Oktober 2006

Danksagung

Der Entstehung dieser Arbeit standen viele Menschen hilfreich zur Seite. Ich möchte Prof. Dr. Joachim W. Schmidt für seine guten Anregungen und fachliche Betreuung danken. Mein besonderer Dank gilt Dr. Hans-Werner Sehring für seine Unterstützung. Er hat sich für die Betreuung dieser Arbeit viel Zeit genommen.

Weiterer Dank gilt Sebastian Bossung für fachliche Unterstützung, hilfreiche Ratschläge und Tipps.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die folgende Arbeit behandelt die Thematik des ***Content Management*** mit besonderem Bezug auf den Aspekt der Erschließung und Wiedergewinnung von Informationsgütern in einem auf Entitäten basierenden System. Dafür wird zunächst eine Einführung in das Thema gegeben und der Aufbau und die Ziele besprochen, bevor dann auf den Inhalt der Arbeit im Detail eingegangen wird.

1.1 Vorwort zum Thema

Die Welt der Rechensysteme drängt nicht erst seit heute in jeglicher denkbaren Form, in alle unsere Lebensräume vor. Durch sie werden uns ständig neue Möglichkeiten und Chancen eröffnet. Analog dazu nimmt aber auch die Menge der damit verbundenen Daten rasant zu. Allein das World Wide Web wächst monatlich um Milliarden von Seiten an, die immer öfter multimediale Daten beinhalten.
Deshalb ist es eine der großen herausfordernden Aufgaben unsere Zeit, mit dieser Flut an Daten umzugehen. Es reicht nicht mehr aus, nur Systeme zum Speichern, Bereitstellen und Verarbeiten von Daten zu haben. Es werden neue innovative Systeme nötig, die über eine intelligente, personalisierte und flexiblere Datenverwaltung und Bereitstellung verfügen, die aber trotzdem einfach handzuhaben sind, um damit neue und effektivere Methoden der Nutzung zu ermöglichen.
An dieser Stelle versucht ein ***Open Dynamic Conceptual Content Management*** (Seh06) anzusetzen. Es erweitert den Klassenbegriff *Content Management* zusätzlich um die Schlagwörter „Offenheit“, „Dynamik“ und „Konzept“. Dahinter steckt die grundlegende Idee, dass strukturierte Daten an sich für uns nur bedingt von Vorteil sind, da sie nicht in der Lage sind, Objekte der realen Welt vollständig zu beschreiben. Um diesem Umstand entgegen zu wirken, bilden Entitäten, die immer aus einem inhaltlichen Teil und einem zugehörigen konzeptuellen Model bestehen, im *Concept-oriented Content Management* (*CCM*) die Objekte der realen Welt ab. Hinzu kommen die Offenheit und die Dynamik, die durch die Möglichkeit von Entitätsbeschreibungen und Veränderung der Entitäten zur Laufzeit umgesetzt sind (Seh04).
Zur Gewährleistung der Effektivität dieses Systems ist es immens wichtig, dass schnell und genau auf die Entitäten und deren Inhalte zugegriffen werden kann.
Also ist eine inhaltsorientierte Suche notwendig, die die Informationswiedergewinnung aus dem System gewährleistet. Doch in großen Systemen treten dabei immer wieder zwei Problemfelder auf, die ***Vagheit*** und die ***Unsicherheit*** (Fer03).
Das eine Problemfeld stellt der Benutzer dar. Der Benutzer kann sein „diffuses“ Informationsbedürfnis nicht präzise und formal ausdrücken. Das andere Problemfeld wird

durch das System selber dargestellt. Ihm fehlen die Kenntnisse über den Inhalt der Informationsobjekte, die auch multimedialer Form sein können. Diese beiden Punkte führen zu fehlenden und fehlerhaften Ergebnissen. Zur Kompensation dieser Mängel dient eine leistungsstarke Suchmaschine. Deswegen wachsen die Effektivität und der Nutzen des Systems mit der Mächtigkeit und Vielseitigkeit der verwendeten Technik der Suchmaschine zur Ermittlung der Informationen.

1.2 Motivation und Ziele

Die Anwendungen, die auf dem *Open Dynamic Conceptual Content Management* Projekt basieren, erfassen eine Menge von multimedialen Daten und Informationen. Um diese zu verwalten und zu repräsentieren, wird eine eigene auf Entitäten basierende Technik verwendet. Diese Entitäten bilden eine untrennbare Einheit von „Inhalt“ und einem konzeptionellen Modell. Im System werden sie durch so genannte ***Assets*** widergespiegelt. Um den konzeptuellen Aspekt gerecht zu werden, beinhaltet ein *Asset* zu den Nutzdaten auch Charakteristika und Beziehungen zu anderen *Assets*. Die Hauptintention dieses Projekts ist es aber, dass das System offen und dynamisch bezüglich dieses Assetsmodells ist. Dazu sind Generatoren notwendig, die auch zur Laufzeit die Komponenten des Systems erstellen können.
Der Zweck der Arbeit ist es, für dieses System eine Vorstudie einer Textsuche in Form eines Entwurfs und seiner Umsetzung zu realisieren. Diese soll ein Konzept beinhalten, wie die Besonderheiten der *Asset*-Technologie wirkungsvoll mit den Eigenschaften einer potenten Suchmaschine verknüpft werden können. Die daraus gewonnenen Erkenntnisse in Bezug auf die Einbindung und Umsetzung der Volltextsuche sollen zur Erstellung eines Generators für dieses System dienen.
Der Entwurf benötigt ein Modell für ein angepasstes Suchen in der Datenschicht. Vereinfacht ausgedrückt, es müssen die in der Datenbank enthaltenden Informationen sinnvoll mit dem *Asset*-basierten System verknüpft werden. Ebenso sollte, durch unterschiedlichen Anfragemöglichkeiten unterstützt, der Systembenutzer schnell zu genauen und sinnvollen Ergebnissen gelangen.
Als Ziel dieser Arbeit gilt die Entwicklung und Erstellung einer prototypischen Komponente für das *Open Dynamic Conceptual Content Management* Projekt, die jenes Konzept umsetzt, dass diese Eigenschaften erfüllt. Dabei sollte der Schwerpunkt der Komponente auf der Erschließung natürlich sprachlicher Texte liegen, um deren Nutzwert durch eine leistungsstarke und vielseitige Suche innerhalb der Projektanwendungen zu steigern.

1.3 Aufbau der Arbeit

Die Realisierung der ***Informationsretrieval-Komponente*** ist in dieser Arbeit in mehrere Teile untergliedert.
Kapitel 2 gibt zunächst einen Überblick über ein *Conceptual Content Management* und

den Aufbau des zugehörigen Systems.
Daraufhin befasst es sich weiter mit dem analytischen Abschnitt und geht dafür zuerst auf die Anforderungen für das *Informationsretrieval* ein. Danach wird der Prozess einer Erschließung von Informationen und deren Rückgewinnung vorgestellt und ein Fazit gezogen. In Kapitel 3 werden die näheren Überlegungen zur Umsetzung der Arbeit anhand eines Entwurfs für eine Komponente ausgeführt und besprochen. Das betrifft sowohl das Design der Modulschnittstellen der Komponente, als auch die Integration der Suchverfahren der verwendeten Suchmaschine.
Die Realisierung des Entwurfs wird im 4. Kapitel behandelt. Dabei handelt es sich um eine rückwärtige Betrachtung der Implementation. Daher werden auch Fragen besprochen wie: „Wo sind Probleme aufgetreten, welche Unterschiede zum geplanten Design gibt es?“
Zum Schluss bietet das 5. Kapitel eine Zusammenfassung der gewonnenen Resultate und gibt einen Ausblick auf zukünftige Herausforderungen und mögliche Anschlussarbeiten.

1.4 Verwendete Notation

Für grafische Darstellungen wird soweit möglich die Unified Modelling Language (UML) in der 2. Version verwendet. Eine Beschreibung ist in (OMG06) zu finden. Zentrale Bestandteile in einigen Abbildungen sind zur besseren Übersicht farblich hinterlegt.

Feste Begriffe und Eigennamen, auf die später verwiesen wird, sind ***fett und kursiv*** dargestellt, wobei deren weitere Verwendung nur *kursiv* formatiert ist. Verweise sind blau ersichtlich gemacht, während Literaturhinweise rot gekennzeichnet sind. Quelltexte und einzelner Programmcode sind in `Courier-Schrift` formatiert. Erklärungen und Hinweise werden durch Sans-Serif-Schrift gekennzeichnet.

Alle hier verwendeten Markennamen und Warenzeichen sind geschützte Begriffe, deren Rechte bei den jeweiligen Firmen liegen und deren Benutzung ausschließlich zu wissenschaftlichen Zwecken und zugunsten des Warenzeicheninhabers verwendet werden. Es besteht keinerlei Absicht, diese Warenzeichen zu verletzen.

2 Eine konzeptionelle Analyse

In diesem Abschnitt soll erarbeitet werden, was benötigt wird, um ein *CCMS* mit einer Volltextsuchmaschine zu erweitern. Dazu wird zunächst eine Einführung in die konzeptionelle Inhaltsverwaltung und deren Architektur gegeben, bevor dann die Anforderungen im Hinblick auf die Integration in die Systemarchitektur und zusätzlich der Funktionsvielfalt einer inhaltlichen Suche erfasst werden können.
Zum Schluss muss eine Umsetzung gefunden werden, mit der sich die Anforderungen erfüllen lassen und diese kritisch hinterfragt und verglichen werden.

2.1 Überblick über die konzeptionelle Inhaltsverwaltung (*CCM*)

Content Management (*CM*)ist ein Sammelbegriff für eine wachsende Anzahl von Organisationsfunktionen, die zusammengefasst den Lebenszyklus digitaler Informationen unterstützen. Entstanden ist das *CM* aus der Notwendigkeit, mit steigendem Datenaufkommen umzugehen. *Open Dynamic Conceptual Content Management* geht einen Schritt weiter und beschränkt sich nicht nur auf die Organisation digitaler Informationen in all ihren Lebenszyklen von der Erzeugung bis zur Ausscheidung aus dem System. Es möchte durch Flexibilität und einer *Content-Concept* orientierten Architektur eine Sicht auf Entitäten darstellen, die die Objekte der realen Welt approximieren und verspricht dadurch vielfältigere und bessere Nutzbarkeit.
Als Inhalt bzw. *Content* werden dabei Text- und Multimedia-Dokumente verstanden. Der konzeptionelle Teil sind Meta-Informationen in Form von Charakteristika und Beziehungen. Diese beiden Einheiten sind untrennbar als *Assets* in einem so genannten ***Inhaltsmodell*** zusammengefasst (Seh04). Abbildung 2.1 verdeutlicht diesen Zusammenhang. Die Charakteristika sind die Eigenschaften, die die Entitäten (umgesetzt durch Assets im System) beschreiben, während die Beziehungen die Verknüpfungen unter ihnen darstellen.

2.1.1 CCM Modellierung

Die Offenheit erhält ein *CCM* durch eine konzeptionelle Modellierungssprache, die es erlaubt, mit *Assets* ein gewünschtes Modell zu gestalten. Daher wird sie auch als Asset-Sprache (*engl.* **Asset Language** (SS03)) bezeichnet. Die Offenheit ist stark mit der Dynamik verknüpft. Damit ein Model offen für Veränderungen und Entwicklung bleibt,

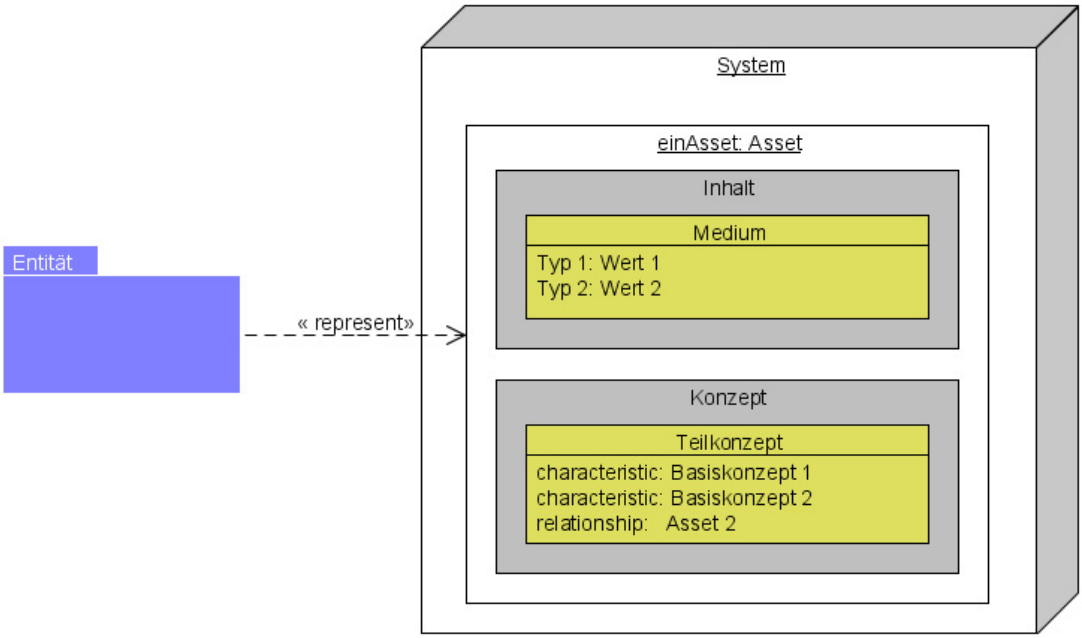

Abbildung 2.1: Genereller Schema von Konzept-Inhalt-Paaren

gibt es ein ***Asset Compiler Framework***, das dynamisch aus den Asset Definitionen die Komponenten für die *CCMS* Architektur generiert.
Ein Beispiel für die *Asset-Modellierungsspache* liefert das Listing 2.1.

Listing 2.1: Asset Definition Language

```
class Person {
        content foto:image
        concept
                characteristic sex:{female,male}
                characteristic name:String
}

class Student refines Person{
        concept
                characteristic martikelnummer:Number
                characteristic Studiengang:String
                relationship university:University*
}

class University {
[...]
}
```

Hier werden anhand der drei Klassen die Konzepte der inhaltlich/konzeptuellen Aufteilung von Assets, sowie die Beziehungsdeklaration und die Strukturbildung durch Verfeinerung konzeptionell aufeinander aufbauender Klassen anschaulich gemacht.
Die Systemarchitektur, bestehend aus Komponenten und Modulen, ist Teil des dynamischen Konzeptes. Durch die beiden Techniken der Dynamik und der Offenheit ist eine flexible, bedarfsgerechte Anpassung des Models möglich. Dadurch entspricht es einer Personalisierung im erweiterten Sinne, da nicht nur das Erscheinungsbild der Nutzungsoberfläche angepasst werden kann, sondern eine komplette Modell Evolution stattfindet.

2.1.2 Systemaufbau aus Komponenten und Modulen

Die *Konzeptorientierte Inhaltsverwaltung* wird realisiert durch ein System, das aus wiederverwendbaren Elementen besteht. Die Wiederverwendbarkeit ist dabei das, was die Architektur dieses Systemes ausmacht. In der Grobarchitektur besteht das System aus Komponenten. Diese unterscheiden sich durch kontextspezifische Kriterien voneinander und repräsentieren durch Kombinationen die Anwendungsdomänen des Systems.

Komponenten wiederum bestehen aus Modulen. Diese realisieren jeweils eine Systemfunktionalität und lassen sich deswegen problemlos wiederverwenden oder austauschen. Sie bilden die Feinarchitektur des Systems und entsprechen den Bausteinen, aus denen das System entsteht. Die Grundlage der Erzeugung der Module für die Komponenten sind die Assetdefinitionen der Anwender. Die Assetdefinitionen bilden das Fundament der Offenheit und der Dynamik des Systems (SS04). In ihnen werden mittels der *Asset Definition Language* die Entitäten des System beschrieben und zur Laufzeit generiert der *Modellcompilers* die beschriebenen Module.
Die Modularisierung des Systems bietet vor alledem den Vorteil einer Kapselung durch die Trennung von Schnittstelle (*Interface*) und Implementierung. Dieser Umstand ermöglicht die Wiederverwendbarkeit einzelner Funktionalitäten. Zusätzlich erfüllen alle Module eines *CCMS* eine einheitliche Schnittstelle zur Verarbeitung von *Assets*. Das fördert die beliebige Einsetzbarkeit weiter.

Davon abgesehen sind die Module in fünf Klassen eingeteilt, die alle eine Basisfunktionalität des System erfüllen. Diese werden in Abbildung 2.2 gezeigt.
Ein ***server module*** Modul entspricht einer Dienstschnittstelle und stellt durch die Bereitstellung einer *API* die Funktionalität komponentenextern zur Verfügung.
Distribution module Module verbinden die Komponenten miteinander, indem sie die Kommunikation zwischen ihnen ermöglichen. Sie bilden die Operationen des einen auf die Operationen des anderen Moduls ab und entsprechen im gewisser Weise Protokollen für Module. ***Mediation module*** Module sind für die Koordination der Anfragen der anderen Module zuständig.
Ein ***mapping module*** Modul entspricht dem Prinzip eines Adapters indem es die Assets in die gewünschten Formen transformiert.

Und ein ***client module*** Modul erlaubt die Zusammenarbeit mit Standardkomponenten (Fremdsystemen) über die von ihnen bereitgestellte Schnittstelle und wird deshalb auch als Interpretationsmodul bezeichnet.

2.2 Anforderungen an die Realisierung

Die Anforderungen für die Realisierung einer umfangreichen „Such-Funktionalität“ in das in Abschnitt 2 vorgestellte System lässt sich in verschiedene logische Abschnitte unterteilen. Im Folgenden wird zunächst die Integration in die *CCMS* Architektur diskutiert, bevor auf die eigentliche Komponentenfunktionalität, die Rückgewinnung von Informationsobjekten, eingegangen wird. Zum Schluss wird noch der Aspekt der Leistungsanalyse hinsichtlich einem Mehrwert gegenüber herkömmlichen Suchverfahren berücksichtigt.

2.2.1 Einpassbarkeit

Ein *CCM* Projekt weist eine mehrschichtige Softwarearchitektur auf, deren Design auf Modifizierbarkeit ausgelegt ist. Dazu gibt es eine strikte Trennung zwischen der Daten-, der Applikations- und der Präsentationsschicht. Die Applikationsschicht wird anhand des Assetmodells vom Modellcompiler erzeugt. Ebenso ist es notwendig, dass die Präsentationsschicht erzeugt wird, aufgrund der Abhängigkeit der ihr zugrunde liegenden Abbildungen vom veränderbaren Assetmodell. Das bedeutet, dass sie durch eine assetbasierte Visualisierung realisiert ist, die der Dynamik durch Assetmanipulationen gerecht wird.
Bei der Datenschicht dagegen können beliebige Standardkomponenten zum Einsatz kommen, auf die dann das konzeptorientierte Inhaltsverwaltungsmodell abgebildet werden kann. An dieser Stelle knüpft ein Information Retrieval Modul an. Es erhält Anfragen aus der Applikationsschicht und leistet dann eine Rückgewinnung der gewünschten Informationen. Für die Rückgewinnung ist eine Standardkomponente zuständig, eine Suchmaschine. Diese liefert aber nur Daten verschiedenen Typs aus der Datenbank ohne direkten Nutzen für ein *Asset* basiertes System. *Assets* sind komplexe Einheiten, bestehend aus einem multimedialen Teil, der die inhaltliche Komponente darstellt und deren Charakteristika und Beziehungen, was den konzeptionellen Teil ausmacht. Diese Daten und Informationen sind nach einen bestimmten Schema in verschiedenen Tabellen der Datenbank des Systems untergebracht. Das macht ein sogenanntes *Mapping-Module*, welches für die richtige Abbildung auf das konzeptorientierte Inhaltsverwaltungsmodell sorgt, erforderlich.
Das *Mapping-Modul* ist eines von fünf verschiedenen Modul Arten, durch die die Struktur des Systems bestimmt wird. Diese Modularisierung erlaubt eine Aufgabenteilung und begünstigt die Wiederverwendbarkeit und Austauschbarkeit der einzelner Teile. Jeder Modul-Typ beschreibt eine Grundfunktionalität und bietet entsprechende Operationen an (siehe dazu Abbildung 2.2).

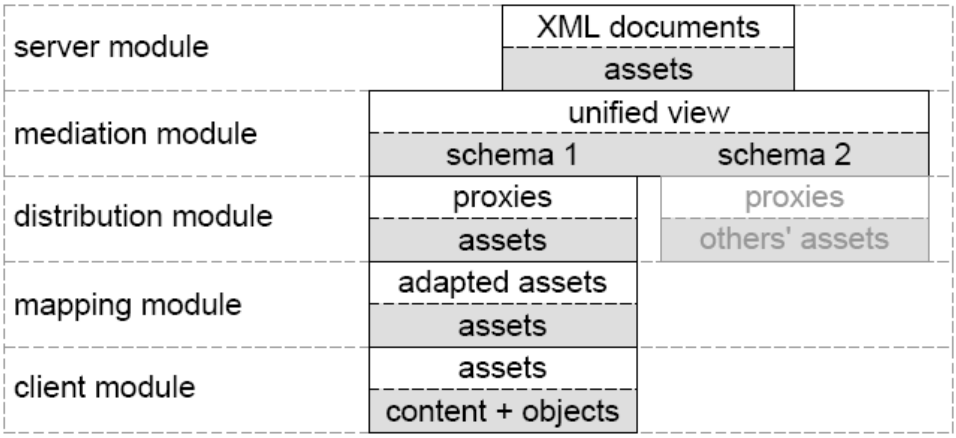

Abbildung 2.2: Modul Arten und hierarchische Übersicht (SS04, Figure 7)

2.2.2 Erweitertes Suchen

Die Aufgabe einer Suchmaschine ist es, den Ort eines bestimmten Objektes oder eine Menge von mehreren Objekten nach bestimmten Kriterien aufzufinden. Für diese Arbeit gibt es eine Vielzahl von verschiedenen Strategien und Ansatzpunkten. Mindestens genauso viele unterschiedliche Möglichkeiten zur Durchführung einer Suche existieren. Die einfachste Variante ist die ***Schlüsselwortsuche***, bei der alle Informationsobjekte nach einem exakten Begriff durchsucht werden. Hierbei lässt sich der Suchraum einfach durch die Verwendung von logischen und mathematischen Operatoren weiter aufspannen. Durch diese können Begriffe einfach kombiniert, in Beziehung zu einander gesetzt oder komplex verknüpft werden, um exaktere Lösungsräume zu erhalten. Eine erweiterte Variante im *Mengentheoretischen Modell* ist das ***Erweiterte Boolesche Retrieval***. Hierbei wird dem 3-Tupel, bestehend aus $T = \{t_i | i = 1, ..., n\}$, das die Menge der Indexterme beschreibt, $Q = \{q_j | j = 1, ...\}$, die Menge aller erlaubten *Queries* und $D = \{d_k | k = 1, ..., m\}$, die Menge der vorliegenden Dokumente, ein weiteres Element $rank(.,.)$ hinzugefügt. Es entspricht einer Rankingfunktion, welche die Ähnlichkeit eines Ergebnisses mit einer Suchanfrage beschreibt (Lee94).

$$4 - Tupel : (T, Q, D, rank(.,.))$$

Durch die Hinzunahme einer *Ranking*-Funktion ist die Sortierung der Objekte des Lösungsraumes nach der jeweiligen Relevanz für die entsprechende Anfrage möglich, was besonders bei großen Ergebnismengen von unschätzbaren Wert sein kann.

Eher eine Erweiterung als eine weitere Alternative ist die ***Fuzzy-Suche*** (Pan86). Darunter sind die Verfahren zusammengefasst, die nicht explizit die angegebenen Suchbegriffe, dafür aber mit diesen stark verwandte Begriffe enthalten. Dies wird auch als ***Inexact match*** bezeichnet. Eine Art ist die ***Phonetische Suche*** , sie ermöglicht es auch, Wörter zu finden, die gleich klingen wie das zu Suchende. Ebenso ist das

Auffinden von Wörtern möglich, die gleich aussehen. Das bedeutet, dass Umgangssprache, sowie Tipp- und Rechtschreibfehler toleriert werden. Zu der Kategorie des *Inexact match* gehören auch die linguistischen Methoden zur Auffindung von Textteilen. Diese sind besonders aufwändig, weil natürliche Sprachen organisch gewachsen sind und nicht etwa entworfen wurden. So entstanden z.B.: *Flexion* (Beugung), *Derivation* (Wortableitung), *Komposition* (Wortzusammensetzung) und Mehrdeutigkeiten, die für die Komplexität der Verfahren sorgen (Lan02) Daraus ergibt sich die Notwendigkeit, sprachliche Umstände in die Erschließung mit einzubeziehen, um den Faktor Mensch zu berücksichtigen. Eine Variante ist das Berücksichtigen der sprachlichen Wurzeln eines Wortes auch ***Stemming*** genannt. Dabei handelt es sich um eine Grundformenreduktion, in der die Wörter auf ihren Wortstamm zurückgeführt werden. Dadurch werden verschiedene morphologische Varianten, wie die Deklination von Nomen und die Konjugation von Verben abgedeckt. Hierdurch ist es möglich innerhalb einer Anfrage sowohl „Fußball“ als auch „Fußbälle“ zu finden oder „gespielt“ und „spielen“.
Eine andere Variante ist das sogenannte ***Thesaurus***. Dabei handelt es sich um ein Wortnetz oder auch Wortschatz *(thesauros, griechisch für Schatz)*, bestehend aus Relationen verschiedener Art unter den Begriffen. Für die Relationsarten von die *Thesauri* gibt es 2 Normen, die DIN 1463-1 und das internationale Äquivalent ISO 2788. (siehe Tabelle 2.1)

DIN 1463-1		**ISO 2788**
BF - Benutzt für	:	UF - Used for
BS - Benutze Synonym	:	USE/SYN Use synonym
OB - Oberbegriff	:	BT - Broader term
UB - Unterbegriff	:	NT - Narrower term
VB - Verwandter Begriff	:	RT - Related term
SB - Spitzenbegriff	:	TT - Top term

Tabelle 2.1: Kürzel und Bezeichnung

Ein *Thesaurus* ermöglicht also unterschiedliche Schreibweisen (Photo/Foto), Synonyme bzw. als gleichbedeutend behandelte Quasi-Synonyme, Abkürzungen, Übersetzungen und Ober- und Unterbegriffe in die Ergebnismenge mit einzubeziehen.
Ebenfalls zum *Inexact Match* gehört das Suchen nach Präfix- und Teilzeichenketten. Hierbei geht es um die Leistungsfähigkeit der Suchmöglichtkeit, was bedeutet, dass neben der Geschwindigkeit auch die Spezifizierbarkeit der Optionen eine Rolle spielt. So z.B. die Anzahl der Zeichen die bei der Bildung eines Indexes für eine Präfixsuche berücksichtigt werden. Durch Benutzung von Platzhaltern kann man dann mit dem Präfix „Auto“ alle zusammengesetzten Wörter finden, die mit „Auto“ beginnen, „Autobahn“, „Autotür“, „Autowäsche“ und so weiter.

Eine andere mögliche Funktion ist das *Pattern Matching*. Damit bezeichnet man das Suchen nach bestimmten Mustern, so z.B: „Finde alle Informationsobjekte die einen *String* enthalten“ oder „Finde Dokumente in denen *Java* und *Entwickler* in einem Satz vorkommen“.

Dann existiert noch eine Methode, die man *Intelligent Match* nennt. Darunter fällt das Suchen nach bestimmten Themengebieten. Es erlaubt Dokumente zu finden, die von einem Thema handeln, in denen der Themenbegriff aber überhaupt nicht erwähnt wird. Ein Beispiel dafür ist ein Artikel über Autos, in dem aber nur Wörter wie Coupe, Cabrio oder Limousine auftauchen. Dazu zählt aber auch das Gegenbeispiel, ein Dokument in dem das Wort „Auto“ erwähnt wird, welches aber über „Reisen“ handelt. Ebenfalls darunter fällt die Fähigkeit, ***Stoppwörter***(Wörter mit nur geringer eigener Bedeutung, die häufig auftreten, wie z. B. Artikel, Präpositionen) zu ignorieren und auch das Können mit nicht-alphanummerischen Zeichen umzugehen, wozu auch das Handhaben von Programmiersprachen wie C++ oder Java gehört.
Für die meisten dieser Verfahren ist es allerdings notwendig, dass die nationalen Unterschiede der Sprachen berücksichtigt werden, also eine breite Sprachunterstützung existiert. Vor allem, weil einige Sprachen, siehe deutsch oder auch dänisch, mehrere akzeptierte Schreibweisen für Wörter haben und dazu noch eine Vielzahl von Sonderzeichen in ihrem Alphabet enthalten.

2.2.3 Suchen in konzeptionellen Inhaltsverwaltungssystemen (*CCMS*)

Aufgrund der auf Assets basierten Architektur konzeptioneller Inhaltsverwaltungssysteme bieten sich, neben einer Volltextsuche noch andere Varianten einer Recherche an, die charakterisierend für ein aus komplexen Objekten bestehendes System sind.
Aus dem Assetschema werden vom Compiler die Klassen generiert, die eine Gruppe von Assets beschreiben. Was allen Objekte gruppenübergreifend gemeinsam ist, ist abgesehen vom Typ ihrer Assetklasse ein Identifikator, der systemweit eindeutig ist. Diese Identifikationsbezeichnung ist eine aus Buchstaben und Zahlen zusammengesetzte Zeichenkette, die die verschiedenen, aber teils identischen Objekte prägnant voneinander zu trennen erlaubt, und stellt deshalb einen primären Parameter für eine Suchanfrage dar.
Genauso erlaubt es die Struktur aus Klassen eine Suchanfrage klassenweit zu formulieren, indem die Assetklasse als ein Übergabeparameter mit in die Anfrage einfließt. Dabei ist aber ein anderes Konzept aus dem objektorientierten Aufbau zu beachten, das *Vererbungsprinzip*. Vererbung kommt dann zum Einsatz, wenn es Klassen gibt, die eine konzeptionelle Basis aufweisen. Die Definition einer neuen Klasse kann gegebenenfalls auf der Definition einer bereits vorhandenen Klasse aufbauen, so dass die neue Klasse die Merkmale der vorhandenen übernimmt. Es entspricht einer Verfeinerung der Oberklasse. Dies bewirkt aber auch, dass bei einer klassenweite Suche eventuelle verfeinerte Klassen berücksichtigt werden müssen und ebenfalls deren, solange bis die letzte Klasse in der vertikalen Struktur mit eingeschlossen ist. Daraus lässt sich ebenfalls eine Volltextsuche erzeugen, indem eine Basisklasse, die den Ausgangspunkt aller existierenden Klassen darstellt, das Argument einer klassenweiten Suche ist.

2.2.4 Performance

Bei der Umsetzung einer Suchmaschine spielt der Ressourcenverbrauch und die Qualität der Ausgabe, wie bei den meisten Komponenten in der Informatik, eine zentrale Rolle dar. Speziell das Zeitverhalten zur Ermittlung von gewünschten Informationen ist ein primärer Faktor für Suchmaschinen. Ausschlaggebend ist besonders die Menge der vorhanden Daten und damit zusammenhängend, aber nicht einhergehend, die Häufigkeit des Vorkommens des zu suchenden Begriffes in eben diesen. Um mit dieser Tatsache umzugehen, gibt es verschiedene Methoden.
Die Suche in einer endlichen Menge wird vor allem dadurch beschleunigt, dass über den Daten ein (Such-)Index (z. B. in Form von n-dimensionalen Bäumen, eines Bitmap-Indizes, oder invertierten Listen (BYa; MS98)) erstellt wird. Dieser unterteilt die Daten bereits beim Einsortieren und kann so die Anzahl der bei einer Suche zu lesenden Datenseiten minimieren. Das erhöht die Geschwindigkeit enorm im Gegensatz zu einem *full-table scan*, bei dem alle Daten durchsucht werden müssen. Bei dieser Prozedur kann man die Anzahl der zu indizierenden und damit zu durchsuchenden Daten weiter reduzieren, indem man sogenannte Stoppwörter von Anfang an ausschließt.
Ebenfalls für einen besseren Durchsatz sorgen spezialisierte Strukturen bei der Datenspeicherung und Erschließung. So existieren unterschiedliche Arten zur physischen Speicherung der Daten (*tablespaces*), die speziell für große oder kleine Datenmengen spezifiziert sind. Diese unterscheiden sich vorzugsweise in der Anzahl der Datendateien und derer Kapazität ihrer Blöcke. Eine Abstraktionsebene höher gibt es die Möglichkeit durch Unterteilte-Tabellen (*partitioned tables*), die Daten sinnvoll aufzuteilen, z.B. nach Datum bei Artikel-Verzeichnissen, um so eine bessere Performance beim Recherchieren zu erreichen.
Der Indizierungsprozess ist dahin spezialisiert, dass für die Erstellung eines Indexes verschiedene Typen existieren, die optimiert sind für unterschiedlichen Aufgabenbereiche und für ihren speziellen Fall jeweils die besten Leistungen versprechen.

2.3 Einführung in *Oracle Text*

Oracle Text ist ist eine Erweiterung der Oracle Datenbank und existiert seit der Version 9i als eigenständiges Element neben *Oracle inter*Media in dem *Oracle* Paket.
Oracle Text ist ein kraftvolle und schnelle Suchmaschine speziell für den Einsatz mit großen textbasierten Dokumenten geeignet (OTF05). Zusätzlich zu der Volltextsuche bietet sie leistungsstarke Funktionen, wie Textklassifikation und *Clustering*, so das neben der Suche mit einzelnen Wortfragmenten auch eine nach Themengebiete orientierte Suche unterstützt wird. Es werden etliche Sprachen unterstützt und eine ausgeklügelte Ranking-Technik verwendet, die für die Qualität der Suchergebnisse sorgt. Außerdem besitzt sie die Fähigkeit, die Suchergebnisse in den unterschiedlichsten Formaten auszugeben und falls gewünscht auch Textstellen hervorzuheben.

2.3.1 Funktionsweise *Oracle Text*

Um eine schnelle und effiziente Suche zu garantieren, arbeitet Oracle mit so genannten Text Indizes. Das sind Verzeichnisse von Schlüsselbegriffen, die bestimmten Dokumenten zugeordnet sind. Diese Schlüsselbegriffe werden als Deskriptoren bezeichnet und sind die Basis zur Erschließung, der in den Dokumenten enthaltenen Sachverhalte.
Um gute Ergebnisse für möglichst viele Einsatzgebiete zu gewährleisten, werden von *Oracle Text* verschiedene Arten der Indexerstellung zur Auswahl gestellt. Das bedeutet, dass die Parametrisierung des Index-Erstellungsaufrufes ausschlaggebend ist, um optimale Suchergebnisse für das existierende Typ-Schema zu erzielen. Eine besonders wichtige Einstellung ist die Angabe des Indextyps. Oracle stellt 4 Indizes zu Auswahl: Einen, der speziell für die Volltextsuche auf großen Dokumenten geeignet ist und dem das Schlüsselwort *CONTEXT* zugeordnet ist.Er benötigt jedoch nach jeder datenveränderten Aktion eine Aktualisierung in Form eines `sync_index` Statements. Ein weiterer Typ wird durch das Schlüsselwort *CTXCAT* bestimmt. Dieser verspricht die beste Leistung für kleine Textdokumente und für Anfragen auf mit unterschiedlichen Typen wie Daten, Währungen oder Beschreibungen gefüllten Tabellen. Zudem unterstützt eine automatische Synchronisierung der Indizes. Außerdem wurde, extra für die Dokumentklassifikation, der Index Typ *CTXRULE* geschaffen. Dieser ermöglicht es, Klartext, HTML oder XML Dokumente anhand ihres Inhalts bestimmten Klassen oder Kategorien zuzuordnen. Zuletzt gibt es noch die Alternative, für seine Informationsobjekte einen Index des Typs *CTXXPATH* zu benutzen, dieser ist speziell für `existsNode()` Anfragen auf XML basierten Spalten optimiert.

Ein Index wird in *Oracle Text* mit der SQL-Anweisung `CREATE INDEX` erstellt, dabei wird die zu indizierende Tabelle mit einer Textspalte parametisiert (Listing 2.2). Außerdem muss neben dem Indextyp auch noch mindst. die Eigenschaft angegeben werden, die den Ort der zu erschließenden Daten spezifiziert.

Listing 2.2: Einfache *SQL* Indexerstellung mit einem *CONTEXT*-Index

```
CREATE INDEX meinIndex ON Tabelle(Textspalte)
indextype IS ctxsys.context
parameters ('DATASTORE CTXSYS.DEFAULT_DATASTORE');
```

Für viele der weiteren Parameter ist es nötig, sie vorher zu konfigurieren, ansonsten werden die Standardeinstellungen ihrer Konfiguration verwendet. Dazu wird zuerst die `ctx_ddl.create_preference` Prozedur benutzt, um die eindeutige Bezeichnung und die Art der Eigenschaft zu definieren. Danach ist es möglich, unter Angabe des vorher festgelegten Bezeichners mit der `ctx_ddl.set_attribute` Anweisung die gewünschten Attribute für die Indexeigenschaft festzulegen.

2.3.2 Der Indizierungsprozess

Beim Indizierungsprozess wird ein *Oracle Text* Index erstellt mit den vorher spezifizieren Parametern und Einstellungen. Bei der Indexierung kommen eine Menge von Elementen mit verschiedenen Aufgaben zum Einsatz. Diese Elemente werden kurz in ihrer Funktionsweise und ihrer Aufgabe erläutert (OTR04). Die Abbildung 2.3 zeigt die Verkettung und die Reihenfolge wie die Elemente beim Indizierungsprozess zum Einsatz kommen.

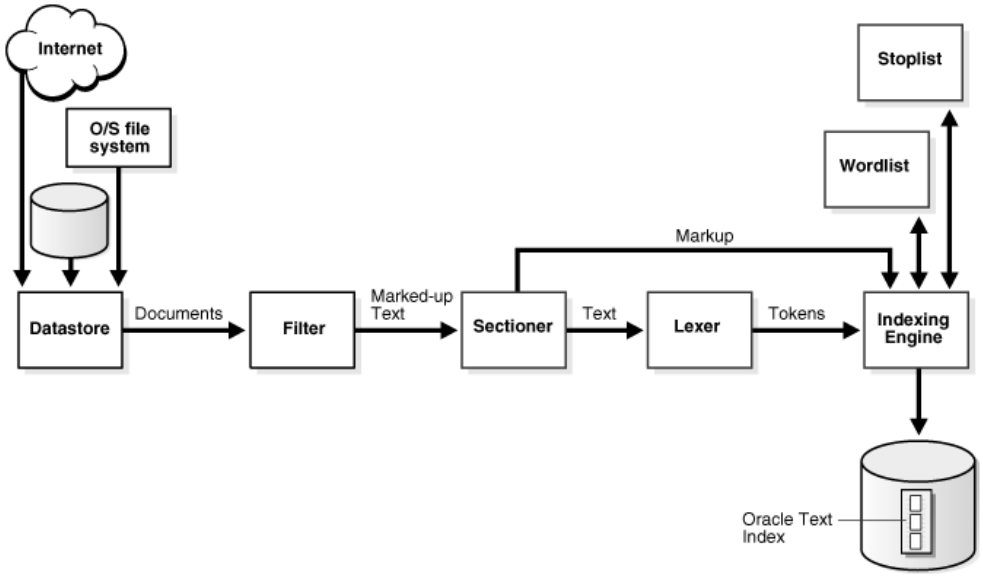

Abbildung 2.3: Indizierungsprozess (OT006, Figure 3-1)

Alle diese Elemente, bis auf den *Sectioner*, lassen sich, oft optional, bei der Indexerstellung durch als Parameter übergebene Eigenschaften konfigurieren.

Datastore Objekt

Das in der logischen Reihenfolge erste Element des Indizierungsprozess ist das *Datastore* Objekt. Dieses liest die Informationsobjekte von dem Ort ein, welcher durch die *Datastore* Eigenschaft spezifiziert worden ist. Dabei können sie aus einer Datenbank, aus dem Internet oder aus einem Betriebssystem stammen. Wichtig ist nur, dass eine textbasierte Tabelle als Referenz existiert, die auf sie verweist. Abbildung 2.4 zeigt diesen Zusammenhang.

Filter Objekt

Als nächstes passieren die Daten das *Filter* Objekt. Was hier passiert hängt von den bestehenden Filter Eigenschaften ab. Es gibt 3 Möglichkeiten:

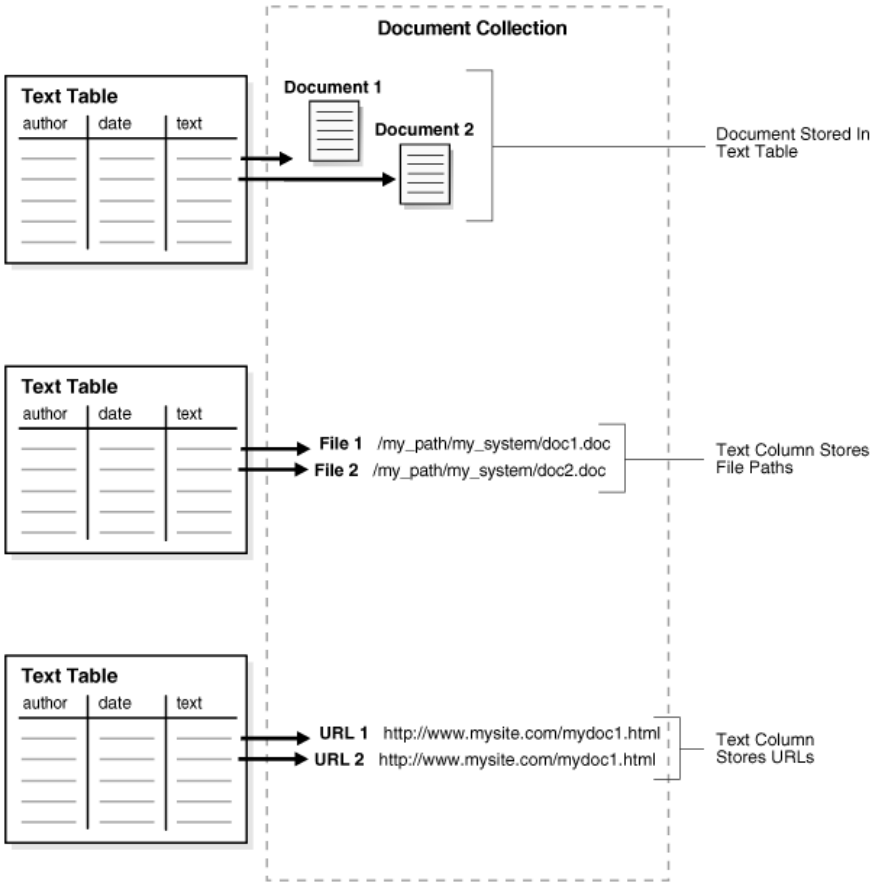

Abbildung 2.4: Arten der Dokumenten Speicherung (OT006, Figure 3-2)

- Die Daten werden gar nicht gefiltert. Bei Informationsobjekten vom Typ Klartext, HTML oder XML bedarf es keiner Filtration. Dazu verwendet man die *NULL_FILTER* Eigenschaft oder man setzt den Wert einer speziellen Formatspalte auf *IGNORE*.

- Informationsobjekte, die in binärer Form vorliegen, werden mit Auszeichnungstext (engl. markup-text) gefiltert. Das dient zur Beschreibung von Daten, indem deren Eigenschaften und Zugehörigkeiten beschrieben bzw. zugeordnet werden. Dies geschieht, wenn die Filter Eigenschaft *INSO_FILTER* gesetzt ist oder der Wert der Formatspalte *BINARY* ist.

- Daten werden von einem nicht kompatiblen Zeichensatz in einem datenbanküblichen Zeichensatz konvertiert. Dafür ist die *CHARSET_FILTER* Filter Eigenschaft zu setzen.

Sectioner Objekt

Nachdem Passieren des *Filter Objekt* werden die ausgezeichneten Daten aufgeteilt in Text und Abschnittsinformationen. Die Abschnittsinformationen geben an, wo syntaktische Abschnitte anfangen und wo sie im Datenstrom enden. Diese Informationen werden direkt an die *Indexing-Engine* weitergegeben. Der Text dagegen wandert zum *Lexer Objekt*

Lexer Objekt

Im *Lexer* wird der Text in die kleinsten Sinn gebenden Einheiten, genannt *Tokens* zerlegt. Diese Segmentierung geschieht meist auf der Wortebene und hängt stark von der gewählten Sprache ab. Zusätzlich ist die Tokenbildung von den Definitionen aus der *Lexer* Konfiguration, wie z.B. ob Groß- und Kleinschreibung eine Rolle spielt oder wie zusammengesetzte Wörter gehandhabt werden sollen, abhängig. Ebenso ist der *Lexer* verantwortlich für die Analyse und *Tokenisierung* des Textes für die Themengebiete-Suche, wenn die *Lexer* Eigenschaft *index_ themes* auf „yes“ gesetzt wird.

Indexing Engine

Zum Schluss endet der Datenstrom in Form von *Tokens* in der *Indexing Engine*. Die erstellt mit den Wortfragmenten einen invertierten Index, der diese den Dokumenten zuordnet, in denen sie enthalten sind. Dabei berücksichtigt sie auch die *STOPLIST*, eine Schwarzeliste, die spezifiziert wird, um einzelne Wörter oder Themenbegriffe bei der Indizierung auszuschließen. Außerdem kann man eine so genannte *WORDLIST* Eigenschaft anlegen, um eine zusätzliche Indexierung von Präfix- und Teilzeichenfolgen zu konfigurieren. Dies erhöht die Geschwindigkeit für die Suche mit Platzhaltern erheblich.

2.3.3 *Oracle Text* Anfragen

Voraussetzung für die Suche mittels Oracle Text ist die zuvor beschriebene Indizierung der zu durchsuchenden Informationsobjekte. Je nach Typ des Indizes ist das Auffinden von Text mit unterschiedlichen Strategien möglich:

- Schlüsselwortsuche
  - Finde alle Dokumente mit einem oder mehreren Schlüsselwörtern
- Kontextsuche
  - Finde alle Dokumente, in denen die Schlüsselwörter in einem bestimmten Kontext auftauchen

- Boolesche Operationen
  - Kombiniere Schlagworte oder Terme mit UND, ODER oder NICHT um exaktere Ergebnisse zu bekommen.
- Linguistische Methoden
  - Finde alle Dokumente über ein Thema
- Pattern Matching
  - Finde Dokumente mit Wörtern, die einen „String“ enthalten

Damit die optimierten Text-Indizes von *Oracle Text* verwendet werden, ist es nötig in der `where`-Klausel des SQL-Statements spezielle parametrisierte Operatoren zu verwenden. Je nachdem, was für ein Indextyp gewählt wurde, gibt es andere Anfrage-Operatoren. Listing 2.3 zeigt eine Standard *Oracle Text SQL* Anfrage, wobei der Platzhalter *Query_ Operator* durch den jeweiligen Suchoperator ersetzt wird und innerhalb der Klammern der Suchterm und weitere Parameter spezifiziert werden können.

Listing 2.3: SQL Query

```
SELECT column FROM table WHERE "query_operator()" > 0;

SELECT name FROM emp
WHERE "CONTAINS(emp,'term',1)" > 0;   /*Context-Indextyp*/
```

Index Typ		Anfrage Operator
CONTEXT	:	CONTAINS
CTXCAT	:	CATSEARCH
CTXRULE	:	MATCHES
CTXXPATH	:	existsNode

Tabelle 2.2: Anfrage Operatoren

2.4 Alternative Realisierungsentwürfe

In diesem Abschnitt wird eine alternative Volltextsuchmaschine vorgestellt, um auch andere Möglichkeiten und Technologien kennen zulernen.
Lucene (Sch06) ist ein in Java implementiertes Open-Source-Framework. Dadurch ist es dem Entwickler im hohen Maße möglich, selbst Einfluss auf die Gestaltung der Suchverfahren zu nehmen. Dafür werden von *Lucene* Klassen zur Implementierung einer Volltextsuche zur Verfügung gestellt. Diese Möglichkeit der Personalisierung macht *Lucene* sehr flexibel.

Um den Suchraum zu definieren arbeitet *Lucene* ebenfalls mit Indizes. Diese erhält es durch Umstrukturierung aller Informationsobjekte in ein eigens erarbeitetes Format, einem *Lucene-Dokument* (LD). LDs wiederum bestehend aus *Lucene*-Feldern,

die einen spezifischen Wert enthalten, der einen speziellen Teil der Eingabedatei umschreibt. Bevor auf Basis dieser LDs eine Index erstellt wird, gibt es auch hier noch die Möglichkeit mittels eines *Analyzers* den Text zu filtern und zu normalisieren. Dadurch werden unwichtige Textfragmente vermieden und eine höhere Performance erreicht.

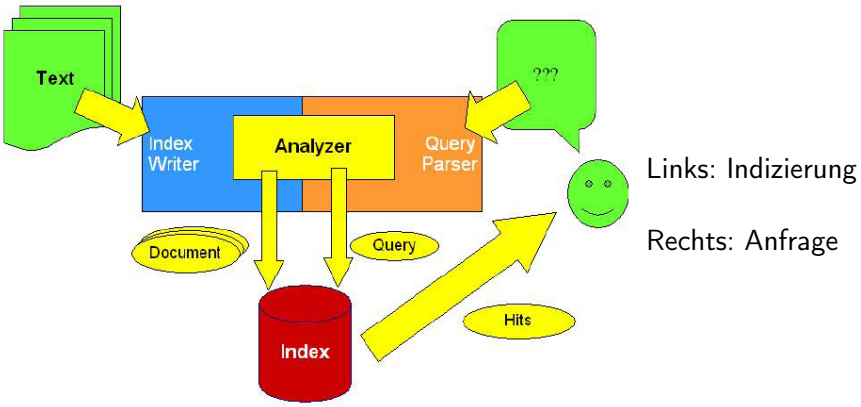

Abbildung 2.5: Gesamtablauf Lucene (Sch06, Abbildung 1)

Die Suche in *Lucene* läuft ähnlich ab. Nachdem der Benutzer eine Anfrage gestellt hat, wird auch diese intern in eine von *Lucene* lesbare *Query* konvertiert. Derselbe *Analyzer* wie bei der Indexerstellung (vorzugsweise) bekommt diese *Query* und normalisiert sie ebenfalls. Mit dem Resultat werden nun die Anfrage aufgelöst und die Ergebnisse gespeichert und wiedergegeben. Auch *Lucene* ist in der Lage, vor der Darstellung der Ergebnisse diese nach Relevanz zu sortieren.

Durch *Lucene* werden gleichermaßen verschiedene Suchmethoden unterstützt (Hei06). So ist eine themenbezogene Suche ebenso ausführbar, wie die Verwendung von *Wildcards* oder booleschen Operatoren in dem Anfrageterm. Außerdem ist das Suchen nur in bestimmten Feldern möglich und eine undeutliche (*Fuzzy*) sowie eine Distanzsuche über einen festgelegten Bereich machbar.

Allerdings ist *Lucene* keine fertige Suchmaschine, sondern stellt lediglich Klassen und Funktionen zur Verfügung. Das bedeutet eine hohen Mehraufwand für Implementierung und Konfiguration bei der Entwicklung. Dazu gehört auch, dass die Darstellung und die Aufbereitung der Resultate im eigenen Realisierungsbereich liegen, dieser Umstand kann sich aufwändig für alle zu unterstützenden Dateiformate, insbesondere ddieer Binären, erweisen. Ein weiterer Nachteil ist, das *Lucene* alle erzeugten Indizes direkt im Dateisystem ablegt. Dadurch wird der gesamte Datenbestand vollkommen von der Persistenz des Dateisystem und der Pfaden abhängig gemacht.
Außerdem sind Algorithmen zur Wortstammreduktion äußert anspruchsvoll und die in *Lucene* vorhandenen *Stemmer*, insbesondere für deutsch, sind nur passabel gelöst.
Darüber hinaus kann *Lucene* die Suchresultate zwar nach Relevanz sortieren, aber dabei handelt es sich nur um einen Sortieralgorithmus, der relativ zu der jeweili-

gen Anfrage ist. Die *Oracle Text* Suchmaschine dagegen besitzt eine ausgeklügelte Ranking-Technik, die eine absolute Bewertung der Dokumente vornimmt. Auch nicht zu vernachlässigen ist die Ambivalenz eines *Open-Source* Projekts. Denn damit ist auch verbunden, dass es keinen professionellen Support, Gewährleistung oder Releasepläne gibt. Das kann zum Beispiel - aktuell im Jahr 2006 - dann wichtig sein, wenn man von Überarbeitungen der Wörterbücher nach Reformen abhängig ist.

2.5 Schwachstellen und Fazit

Mit der Wahl, *Oracle Text* als externes Fremdsystem für die Textsuche mit einem *CCMS* zu nehmen, ergeben sich dennoch ein paar potenzielle Nachteile.
Dies betrifft zuerst *Oracle Text* selber. Eine laufende Indizierung lässt sich nicht abbrechen oder pausieren. Dies hat zur Folge, das bei großen Datenmengen sehr viele Ressourcen für eine lange Zeit gebunden sind. Zwar unterstützt *Oracle Text* auch paralleles Indizieren, aber da es sich dabei um einen sehr E/A intensiven Prozess handelt bringt dies nur im Mehrprozessorbetrieb eine Performancesteigerung. Andererseits ist das Erstellen *eines* Indexes mit mehreren CPUs nicht möglich.
Des Weitern ist keine automatische Erkennung von Duplikaten vorgesehen, was das Speichern identischer Informationsobjekte vermeiden würde.
Ein Nachteil für Benutzer deutscher Texte ist, dass standardmäßig keine *Knowledge Base* in deutsch vorhanden ist. Diese ist für die Themensuche (*Thesaurus*) nötig und von Anfang an nur für Englisch und Französisch eingerichtet. Außerdem ist *Oracle Text* nicht selbst ständig in der Lage, Information aus Metadaten von Objekten wie Dokumenten oder Bildern zu verarbeiten. Dies würde eventuellen Mehraufwand vermeiden und neue Möglichkeiten zur Suche eröffnen.

Eine andere Problematik betrifft das System einer konzeptionellen Inhaltsverwaltung. Durch das System wird anhand von Asset Definitionen ein festes Datenbank Schema generiert. Das bedeutet, dass die komplette Daten Struktur bereits vorhanden und nicht erweiterbar ist. Dieses schränkt die Einsatzmöglichkeiten der Suchmaschine im Verbund mit einem *CCMS* ein. Für viele von Oracle unterstütze Features werden zusätzliche Datenwerte (Spalten in Tabellen) zur Informationsgewinnung benötigt. Dies können Format- oder Sprachspalten sein, die beim Indizieren der jeweils in der Zeile enthalten Informationen für die Art ihrer Erschließung ausschlaggebend sind. Für mehrsprachige Tabellen kann man ein Multi-*Lexer* Objekt erstellen, das auf eben diese Sprachspalte zugreift und danach die Tokens erstellt. Zwar ist der Einsatz einer *Multi-Lexers* theoretisch auch ohne extra Spalte möglich, dann beruht die Spracherkennung allerdings nur auf Annahmen und ist weniger sinnvoll.

Eine andere Problematik ergibt sich aus der mangelnden Unterstützung der Spalten, die keinen textbasierten Typ aufweisen. So ist es durchaus gewollt, in einer Volltextsuche mittels Angaben von Jahreszahlen oder anderer Daten die Lösungsmenge einzugrenzen. Das aber bedarf einer Einbeziehung von entsprechenden Spalten in den Index, die

einen Zeitstempel enthalten. Dieses ist zwar grundsätzlich durch die Umwandlung der Datenwerte mittels einer speziellen Funktion (`TO_CHAR()`) möglich, aber nur in den Tabellen, die zusätzlich noch eine Spalte in einem Textformat enthalten. Der Grund dafür ist, dass bei der Erzeugung eines Indexes eben genau so eine Spalte benötigt wird, was wieder zu der Unveränderbarkeit des Schemas führt.

Trotz alledem ist *Oracle Text* eine geeignete Suchmaschine für ein *CCM*. Sie bietet die kompletteste Ausstattung mit der besten Performance und ist sowohl von der physikalischen Speicherung der Daten als auch von der gewählten Plattform unabhängig. *Oracle Text* unterstützt einer breite Palette von Text-Formaten, genauso wie etliche Multimedia-Formate. Außerdem arbeitet *Oracle Text*, ebenso wie *Lucene* hervorragend mit auf Java basierenden Anwendungen zusammen und findet mit einer *Oracle* Datenbank für ein *CCM* Projekt eine Komponente aus dem gleichen Hause vor. Ausschlaggebend für *Oracle Text* sind vor allem die vielseitigen Möglichkeiten der unscharfen Suche, die durch ihre qualitative Umsetzung eine Bereicherung darstellen.

3 Architekturentwurf

Die Umsetzung einer Informationretrieval Funktionalität für ein konzeptorientiertes Inhaltsverwaltungssystem erfordert neben der Implementierung der *Oracle Text* spezifischen Methoden für die Suche vor allem auch die Integration in die Architektur eines *CCM* Systems. Der Aufbau des Systems wird ausführlich im Groben und im Feinem in Kapitel 2.1.2 vorgestellt. Im Folgenden wird nun auf die Gestaltung der Umsetzung im Besonderen eingegangen.

3.1 Designkonzept

Aus der oben gegebenen Architekturbeschreibung eines CCMs ergibt sich folgendes Design für die Gestaltung einer Informationretrieval-Komponente:
Zur Umsetzung der Komponente werden nur zwei Module benötigt. Dieses entspricht gleichermaßen der minimalen Anzahl an Modulen für die Realisierung einer Komponente.
Zum einem für den Zugriff auf *Oracle Text* als Fremdsystem ein *client module* und zum anderen zur Umsetzung der Funktionalität und zur Transformation der Assets ein *mapping module*. Den Aufbau der Komponente zeigt Abbildung 3.1 anhand einer Realisierung für ein Projekt der *WEL*, „Geschichte der Kunstgeschichte im Nationalsozialismus“, kurz *GKNS*.

Definiert wird die Komponente in einer Komponentenkonfigurationsdatei. Diese liegt im XML Format vor und legt sowohl die Struktur der Komponente, als auch eventuelle Parameter für die einzelnen Module fest.

3.2 Modul Interaktion

Die Aufgabe des *mapping module* Moduls ist es, die Funktionalität von Oracle Text umzusetzen. Dazu gehört das Wiederauffinden der aus den Assets stammenden Informationen in der Datenbank. Diese liegen dort für jedes Asset in den Spalten einiger Tabellen in unterschiedlichen Formaten vor. Technisch folgt daraus die Notwendigkeit einer Datenkonversion, auf logischer Ebene aber erfolgt eine Abbildung auf Assets. Dafür gibt es das *client module* Modul, welches einerseits den Zugriff auf die Datenbank über die JDBC-Schnittstelle erlaubt und andererseits als „*Wrapper*“ während der Kommunikation agiert. Außerdem das *mapping modul* Modul selbst, das zusätzlich als

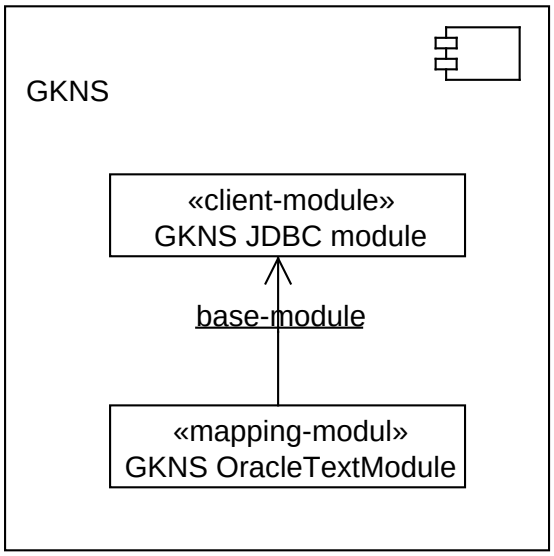

Abbildung 3.1: Aufbau der IR Komponente

Adapter dient. Diese beiden Module sind über eine einheitlichen Schnittstellen (das *client module* Modul hat nur eine) miteinander verbunden.

3.2.1 Schnittstellen für Module

Module werden durch zwei Schnittstellen mit Modulen der über- und untergeordneten Schichten verbunden. Zur flexibleren Kombinierbarkeit erfüllen sie, wie alle Module, eine einheitliche Schnittstelle. Diese Schnittstelle bietet generische Operationen an, deren Parameter aus Assets bestehen.
Sie lassen sie in vier Funktionsklassen einteilen.

- Auffinden (lookfor)
- Verändern (modify)
- Erstellen (create)
- Entfernen (delete)

Wie die Operationen genutzt werden, hängt von dem Umgang mit der als Parameter gelieferten Assets durch das jeweilige Modul ab. Aufgabe des *mapping module* ist es, die IR-Funktionalität in den Operationen zu implementieren. Dazu gehört zum Beispiel, dass bei einer „modify“ Operation die veränderten Daten auch mit den Index der Suchmaschine synchronisiert werden oder dass bei einer Suchanfrage die Ergebnisse mittels der vielfältigen Suchmechanismen aus den erschlossenen Informationen der

Suchmaschine gewonnen werden.
Beim *client module* dagegen erfüllt nur eine ihrer Schnittstellen die einheitliche Modulschnittstelle. Die andere dient zur Nutzung der Datenbank über die von ihr angebotene JDBC-Schnittstelle für Java und relationale Datenbanken. Allerdings ermöglicht genau das Vorhandensein dieser einen standardisierten Schnittstelle, dass das *client module* zusätzlich als „*wrapper*“ für die vom *mapping module* beanspruchten Leistungen der Datenbank dient.

3.2.2 Ablauforganisation

Die beiden Module sind trotz ihrer Eigenständigkeit eng miteinander verknüpft. Ihr Verhältnis zueinander ist so gestaltet, dass das *mapping module*, welches das Prinzip eines Adapter verwirklicht, eine „auf ihm basierende“ Beziehung zum *client module* hat. Das *client module* übernimmt dabei die Aufgabe der Kommunikation mit der Datenbank. Demnach werden die Grundoperationen beim *mapping module* von der Funkion her nicht eigens implementiert, sondern an das *client moduel* weiter delegiert. Es wird allerdings registriert, dass der Zustand eines oder mehrer Assets sich verändert hat und darauf reagiert.
Diesen Umstand veranschaulicht Abbildung 3.2.

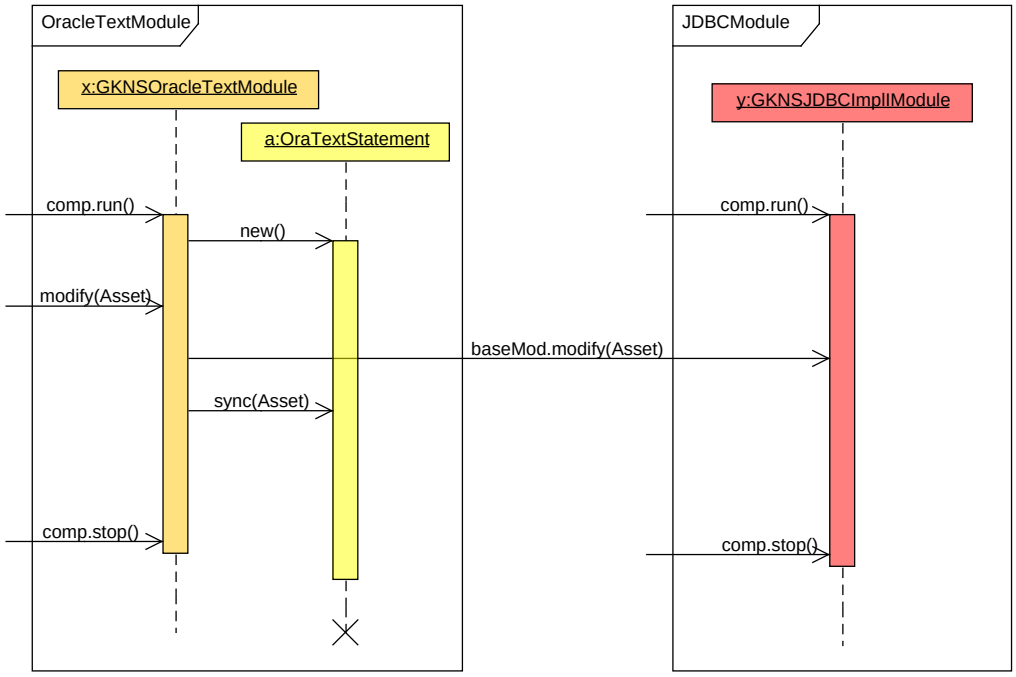

Abbildung 3.2: Ablauf einer Änderung von Asset Werten

Es zeigt die Ausführung einer Änderungsoperation in Gestalt der Weiterdelegation der

eigentlichen Operationsausführung an das *client module* (JDBCModule) und der Reaktion des *mapping module* (OralceTextModule) durch eine `sync(Asset)` Anweisung. Eine besondere Variante sind die Operationen zum „Auffinden“, bei denen es zu einer Umwandlung der Assets kommt. Dies geschieht durch eine Zerlegung der vollständigen Assets in ihre Bestandteile und eine vorübergehende Reduktion auf ihre Identifikatoren. Die Identifikatoren sind die „IDs“, durch die die Assets eindeutig kennzeichnet sind. Dieses ist sinnvoll, weil für die Funktionalität der Suche nur die Identität der Informationen in Abhängigkeit von den Assets, nicht aber die gesamte Gestalt der Assets von Bedeutung ist.

3.3 Klassenentwurf

Der Entwurf sieht vor, dass das *mapping module* nicht wie die meisten Module durch den Modellcompiler generiert wird. Das bedeutet, dass die Schemainformationen aus den Assetdefinitionen intern nicht automatisch bekannt sind.
Um trotzdem die Generik zu bewahren und gegenüber des Schemas unabhängiger zu sein, wird für das *mapping module* eine abstrakte Oberklasse eingerichtet, in der die schemaspezifischen Methoden lediglich definiert und nicht implementiert werden. Dieses hat den Vorteil, dass die abstrakte Klasse alle Attribute/Methoden der Modulschnittstelle als gemeinsame Schnittmenge aller konkreten schemaspezifischen Unterklassen enthält. Diese vererbt sie an eine konkrete Klasse, die dann die jeweiligen Informationen ihres Schemas mit den dazugehörigen Assets verarbeiten muss.
Abbildung 3.3 zeigt diesen Zusammenhang und den Bezug der Schemakonkretisierung anhand eines Entwurfs für das *GKNS* Projekt.
Auf die Bedeutung der Klassen für den Aspekt der Schemakonkretisierung wird im darauffolgenden Abschnitt 3.3.1 eingegangen.

3.3.1 Klassen der Verwaltungsebene

Wie die Abbildung 3.3 zeigt, werden durch die Modulschnittstelle zusätzlich zu den Asset-Management-Operationen noch einige Funktionen zur Komponentensteuerung definiert.

- Voreinstellen (init)
- Starten (start)
- Beenden (stop)

Der Zweck der Methoden besteht darin, dass beim Start der Komponente, zum Bespiel durch den Aufruf „`component.run()`“ zuerst die „`init()`“ Methode von der implementierenden Klasse und dann erst der Aufruf von „`start()`“ erfolgt.
Der Vorteil ist, dass zunächst einmal Parameter eingelesen werden können, z.B. für die Verbindung zu der Datenbank. Darüber hinaus besteht die Notwendigkeit, dass dem

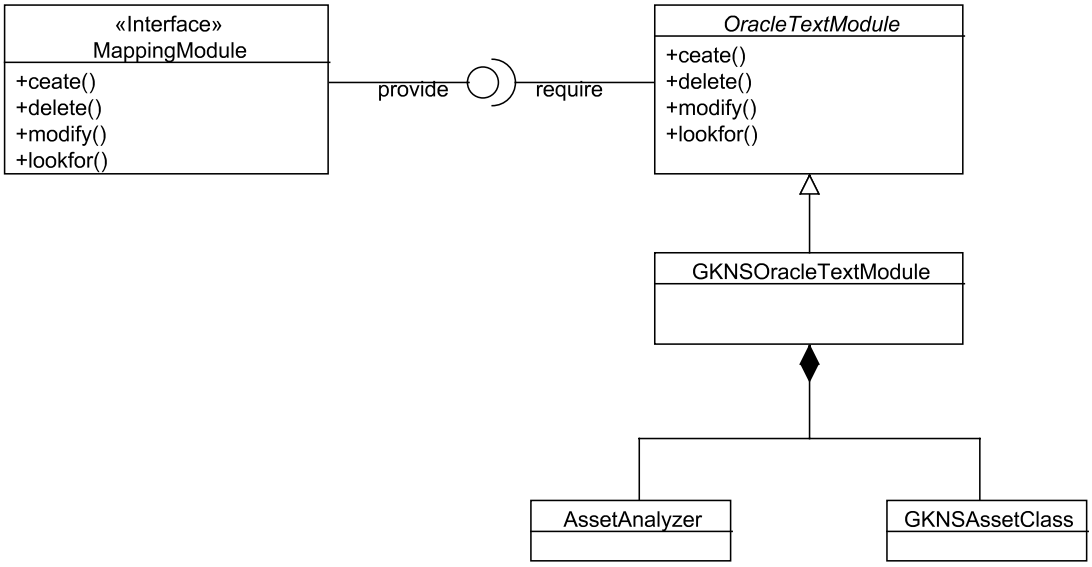

Abbildung 3.3: Klassenübersicht anhand des GKNS Projektes

von der Klasse instanzierten Objekt ihre Komponente bekanntgemacht werden kann. Erst danach, durch den Aufruf der „`start()`“ Methode wird überhaupt versucht, eine Verbindung zu der Datenbank herzustellen. Außerdem ergibt sich hier die Möglichkeit, den Status der Daten in der Datenbank für die Suchmaschine zu überprüfen und gegebenenfalls Aktualisierungen durchzuführen.
Die „`stop()`“ Methode wurde eingerichtet, um ein ordnungsgemäßes Beenden der Datenbank Sitzung, sowie das eventuelle Sichern von Informationen zu gewährleisten.

Des Weiteren verdeutlicht die Abbildung 3.3, wie mit dem Fehlen der Assetschema spezifischen Informationen umgegangen wird. Es existieren dafür zwei Klassen, die unterschiedliche Strategien implementieren, die je nach Bedarf angewendet werden.

1. **GKNSAssetClass**
   Diese Klasse umschließt die Informationen der in dem jeweiligen Schema vorkommenden Assetklassen. Außerdem ist sie für die Verarbeitung und Aufbereitung der klassennahen Daten zuständig, vor allem vor dem Hintergrund der vertikalen Strukturen von Assetklassen. Ihre Informationen über die definierten Assetklassen bezieht sie aus einer Schemadatei, die speziell für dieses Schema angelegt wurde.

2. **AssetAnalyzer**
   Die Klasse AssetAnalyzer analysiert gegebene Assets und bestimmt einerseits ihre Assetklasse. Andereseits erstellt sie aus den aus zerlegten Assets stammenden Daten Suchanfragen für Ähnlichkeitssuchen.

3.3.2 Klassen fürs Informationsretrieval

Um die Funktionen der Suchmaschine zu realisieren, sind einige extra Klassen notwendig, die sich mit der Kommunikation und der Verarbeitung der datenbanknahen Daten befassen und die die datenbankinterne Struktur innerhalb des Systems abbilden.
Auf der einen Seite ist eine Klasse vorhanden, die alle Methoden implementiert, die mittels Oracle eigener Typen den Austausch von Daten und Informationen regeln. Diese Klasse arbeitet daher nicht auf der Assetebene, sondern kennt lediglich einzelne notwendige Werte ihrer Attribute.
Die andere Seite stellen Klassen innerhalb des Modules dar, die die einzelnen Objekte des Indizierungsprozesses widerspiegeln. Dies ist deswegen sinnvoll, weil diese Elemente in der Datenbank ebenfalls Objekte darstellen, die unabhängig erstellt und konfiguriert werden müssen, bevor sie beim Indizieren benutzt werden können. Dies ist nur dann nicht notwendig, wenn lediglich die Standardeinstellungen der jeweiligen Elemente beim Erstellen eines Index verwendet werden.
In der Abbildung 3.4 werden die Klassen, die die Elemente des *Indizierungsprozesses* realisieren, und deren Zusammenhang mit der *„Statement“ Klasse* dargestellt. Außerdem ist die Beziehung zu der *DBResult* Klasse illustriert, die bei der Verarbeitung der Anfragetreffer eine entschiedene Rolle spielt.

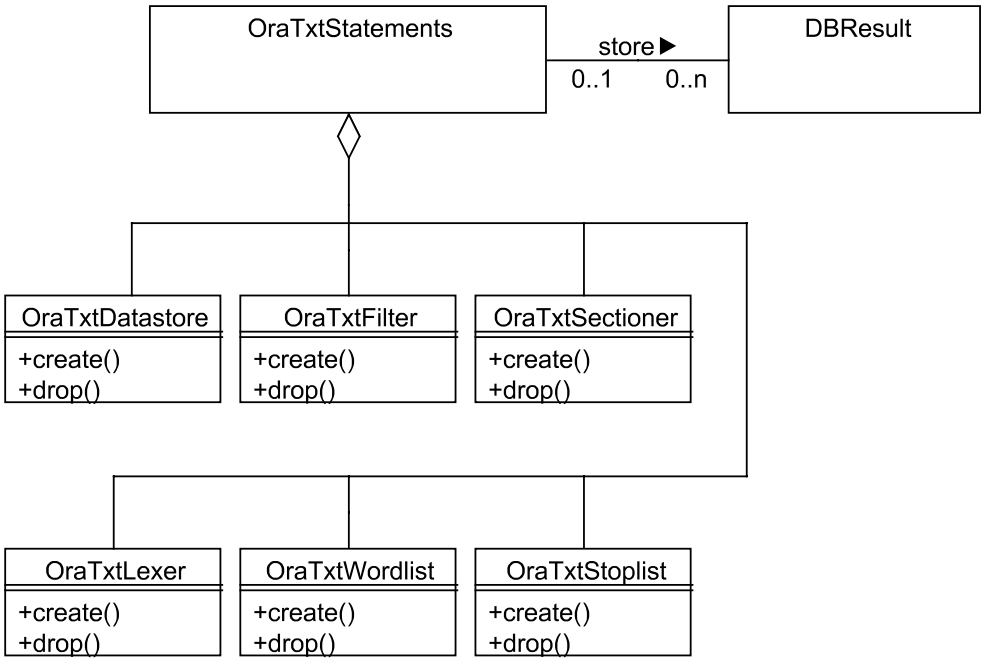

Abbildung 3.4: Oracle Text Klassen

Besonderes Augenmerk verdient die Klasse *OraTxtDatastore*. Das *Datastore* Element legt den Ort fest, an dem die Daten liegen, die indiziert werden sollen. Das kann neben

der Datenbank auch im Internet oder in Dateien auf einem physischen Datenträger sein. In diesem Zusammenhang ist es nur nötig, dass es eine Text-Tabelle gibt, die die Daten entweder selber hält oder auf sie verweist. Beispielweise kann dieses in Form von *URLs* oder Pfadangaben geschehen.
Bei der Möglichkeit der direkten Speicherung in der Datenbank existiert aber noch eine zusätzliche Variante, die es erlaubt mehrere selbst definierte Spalten in einem einzigen Index mit aufzunehmen. Das können dann, unter der Voraussetzung dass zumindest eine Spalte mit Werten eines Texttyps existiert, auch Spalten anderer vorgegebener Formate sein. Diese müssen nur zuvor mit Hilfe der Oracle SQL Funktion ,,`TO_CHAR()`‘‘ konvertiert werden. Diese Variante heißt *MULTI_ COLUMN_ DATASTORE* und muss wie die anderen explizit erstellt und konfiguriert werden.
Das ist die Aufgabe der Klasse *OraTxtDatastore*. Bei jeder Indexerstellung wird ein eigenes *MULTI_ COLUMN_ DATASTORE* Objekt benötigt. Demnach ist es notwendig für alle Tabellen solche Objekte zu erstellen und deren Attribute zu ermitteln und zu setzen. Dabei wird durch die Attribute neben den zu indizierenden Spalten auch noch festgelegt, welche der Spalten gefiltert werden sollen.

Zusätzlich muss die Klasse auch entscheiden, welche Tabellen überhaupt einen Index für eine Textsuche erhalten können. Gründe für eine Nichtberücksichtigung wären zum Beispiel das bereits erwähnte Fehlen von Textspalten. Eine andere Tätigkeit ist das Umgehen mit den *Datastore* Objekten. Wird versucht ein *MULTI_ COLUMN_ DATASTORE* Objekt mit demselben Namen zu erstellen wie ein bereits Existierendes, kommt es zu einem Fehler, genauso wie wenn ein Objekt verwendet wird, dessen Attribute nicht zu der angegebenen Tabelle passen.

3.4 Operationen zur informellen Rückgewinnung

Die Konzeption der Operationen aus der Modulschnittstelle zum Auffinden von Informationen spielt eine zentrale Rolle. Sie bildet das Herzstück einer Informationsretrieval Komponente. Während andere Operationen aus der einheitlichen Schnittstelle vorwiegend nur die Anweisungen an das Basismodul weiter delegieren und selbst nur mit einer Aktualisierung der Indizes auf die geänderten Daten reagieren, setzt die Methode zum „Auffinden„ einen breiten Funktionsumfang um.

3.4.1 Die Retrieval-Operation auf Assetebene

Es handelt sich dabei um die `lookfor()` Methode, die durch unterschiedliche Parametrisierung verschiedene Mechanismen der informellen Rückgewinnung unterstützt. Die Ergebnisse werden dabei immer in Form von Assets zurückgeliefert.
Die einfachste Form ist das Suchen nach einem konkreten Asset. Dieses wird anhand seines Identifikators, der der `lookfor()` Methode übergeben wird, exakt identifiziert. Die gegensätzliche Strategie ist die Volltextsuche. Dabei wird mittels der

`lookfor(Anfragetext)` Implementierung die gesamte Datenbank nach Assets durchsucht, die dem Anfragetext entsprechen. Die Begriffe, die den Suchraum definieren, werden in diesem Fall tolerant interpretiert, sodass auch Unterschiede im Kasus zu Treffern bei Assets führen. Dazwischen existieren weitere Möglichkeiten, die zur Verfeinerung der Suchergebnisse dienen.
Eine Alternative ist das Durchsuchen der Assets lediglich bestimmter Assetklassen nach den gewünschten Informationen. Diese Art der Recherche schränkt den Suchraum auf die angegebene Assetklasse und auf alle auf ihr konzeptionell aufbauenden Klassen ein.
Eine weitere Möglichkeit bietet die `lookfor(exampleAsset)` Methode. Bei dieser Abwandlung wird die Ergebnismenge bestimmt durch die Werte der konzeptionellen und inhaltlichen Attribute eines Musterassets (*Query by Example*). Anders ausgedrückt, man kann anhand eines Beispiels weitere Objekte ausfindig machen, die dem ähneln. Eine Variation dieser Variante erlaubt zusätzlich auch mit mehreren Musterassets zu suchen.
Dies wird so erreicht, dass die Inhalte der Attribute über Disjunktionen mit den anderen Attributwerten der Assets kombiniert und falls vorhanden durch Konjunktion mit den weiteren Disjunktionstermen verknüpft werden. Die daraus entstehende ***Konjunktive Normalform*** bildet dann die Anfrage. Hier ein Beispiel für die Generierung einer Anfrage aus 3 Assets, die jeweils auch 3 Attribute haben:

$$Anfrage = (a_2 \vee a_1 \vee a_0) \wedge (a_2 \vee a_1 \vee a_0) \wedge (a_2 \vee a_1 \vee a_0)$$

Anders funktioniert die Einschränkung des Lösungsraumes über Nebenbedingungen, sogenannten *QueryConstraints*. Damit sind Bedingungen, ausgedrückt durch relationale Operatoren wie Vergleichoperatoren und Ähnlichkeitsoperatoren, gemeint, die für Charakteristika und Relationen definiert sind.
Diese Variante der `lookfor()` Methoden ist ein besonderer Fall. Denn lediglich ein spezieller Teil der Lösungsmenge wird durch das *IR-Modul* aufgelöst, der Rest wird durch Weiterleitung an das *JBDC-Modul* gewonnen. Ausschließlich der Teil einer Suchanfrage, bei dem Bedingungen für Charakteristika mit dem Ähnlichkeitsoperator (*similar*) auftreten, wird durch die Suchmaschine geleistet.
Das bedeutet, dass wenn abgesehen vom *Similar-Operator* noch andere Operatoren vorkommen, es zu einer heterogener Lösungsmenge kommt, die sowohl aus Teilen vom *mapping module* als auch vom *client module* besteht.

3.4.2 Die Retrieval-Operation auf Datenebene

Alle vorhergehenden Varianten der `lookfor()` Operation sind Realisierungen auf der Assetebene. Das bedeutet, dass sie nur mit dem Konzept Asset und seinen Elementen in Berührung geraten. Die Gewinnung und Verarbeitung der „Rohinformationen“ aus der Datenbank wird dagegen von Operationen aus der *OraTextStatement* Klasse geleistet.
Das sind die Methoden, die die Tabellen eines *CCMSs* nach den Textfragmenten durchsuchen, die einer gegebenen Anfrage entsprechen (SMW06). Dafür wird aber nicht jede

Tabelle wie bei der strukturierten Suche mit *SQL* abgefragt, sondern es werden die zuvor erstellten Indizes bei der Formulierung der Anfrage verwendet. Diese Indizes bauen allerdings wieder auf mindestens einer Spalte der Tabellen auf, sodass es ebenfalls mindestens so viele Indizes wie die Menge der zu durchsuchenden Tabellen gibt. Bei einer Volltextabfrage auf den gesamten Datenbestand werden dementsprechend so viele Anfragen an die Datenbank gemacht, wie es Indizes gibt, und die Lösungsmenge wird aus der Menge der zurückgegebenen Ergebnisse gewonnen.
Handelt es sich um eine unscharfe Suchanfrage, wird jede dieser Anfragen durch mehrere Suchoperatoren erweitert, um eine vielseitigere Lösungsmenge zu erhalten. Umgesetzt wird dies durch eine Kombination der mit dem Anfrageterm verknüpften Operatoren im `CONTAINS` Teil der `WHERE` Klausel (Listing 3.1) in jeder einzelnen Anfrage.

Listing 3.1: SQL Beispiel einer Anfrage mit mehreren Operatoren

```
SELECT SCORE(1) AS score, cocoma_id
FROM Tabellenname
WHERE CONTAINS(Indexspalte,'fuzzy(Anfrageterm, 60, 6,weight),
$Anfrageterm, !Anfrageterm ', 1) > 0
ORDER BY SCORE(1) DESC;
```

Diese aus mehreren Suchmethoden zusammengesetzte Anfrage wird dann auf jeden Index angewendet und die Ergebnismengen verknüpft. Dabei werden doppelte Treffer aussortiert.
Die Suchoperatoren, die bei der unscharfen Suche verwendet werden, sind der *Stemming*-Operator (*stem ($)*), der Begriffe liefert, die dieselbe linguistische Wurzel haben, der *soundex (!)* Operator, der auch Wörter einschließt, die „ähnlich klingen“ und der *Fuzzy*-Operator, der auch Treffer erzeugt, die gleichartig geschrieben werden.
Bei der Auswertung der Ergebnisse werden allerdings nicht die übereinstimmenden Treffer aus den Tabellen in die Lösungsmenge aufgenommen, sondern stattdessen nur die Bezeichner, die diese eindeutig einem Asset zuordnen. So wird genaugenommen nicht nach den Textfragmenten aus dem Anfrageterm recherchiert, sondern es werden die Assets (in Form von Identifikationsschlüsseln) gesucht, die solche Treffer in ihren Attributen aufweisen. Die Speicherung findet dann durch Objekte der Klasse *DBResult* statt, die wiederum in Listen gepackt werden. Außerdem werden zusätzlich zu den IDs, die die Lösungsmenge bilden, *Ranking*-Werte als Ordnungskennziffern in Attributen der *DBResult* Instanzen gesammelt. Diese werden durch die Suchmaschine ermittelt und dienen der Relevanzbewertung des entsprechenden Assets. Der Vorteil der *Oracle Text Ranking*-Technik besteht in diesem Zusammenhang darin, dass ein *Scoring* jedes einzelnen Ergebnisses vorgenommen wird. Das bedeutet, dass die Suchmaschine ein absolutes Maß für die Bewertung und Einordnung liefert. So ist es dadurch erst möglich, die Relevanzbewertungen der unterschiedlichen Anfragen, wie sie auch für einzelne Assets nötig sind, in Relation zueinander zu setzten und eine gesamte Ordnung auf den Ergebnissen herzustellen.

4 Softwaretechnische Umsetzung

Als softwaretechnische Umsetzung des im vorangegangenen Kapitel geschilderten Designs wurde im Rahmen dieser Bachelor Arbeit eine Komponente für eine Textsuche konzipiert und in Java implementiert. Diese Implementierung wurde konkret für das Projekt ‘Geschichte der Kunstgeschichte im Nationalsozialismus“ der WEL entwickelt und beruht auf dem damit verbundenen GKNS-Assetmodell.
Die Komponente besteht aus zwei Modulen, die verschiedene Aufgaben übernehmen. Dennoch ist die Beziehung zwischen ihnen so gestaltet, dass das eine auf dem anderen basiert. Das Basismodul wird dabei von einem Generator erzeugt und ist nicht Teil der Implementierung gewesen. Daher wird im folgenden Abschnitt nur das *mapping modul*, das die eigentliche Funktionalität umsetzt, erläutert und beschrieben. Es benutzt die Technologie von *Oracle Text*, um die Inhalte der Assets zu erschließen und die Operationen der Modulschnittstelle umzusetzen, und wird aus diesem Grund als ***OracleTextModul*** bezeichnet.

4.1 Einführung in den Aufbau der Klassen

Bei der Realisierung des *OracleTextModule* Moduls wurden die relevanten Verfahren, die nötig sind, um eine große Menge an Informationen inhaltlich zu erschließen und eine anschließende Wiedergewinnung dieser zu ermöglichen, mit wenigen Klassen in Java umgesetzt. Zur Kommunikation mit der *Oracle* Datenbank wird von ihnen die *JDBC*-Schnittstelle genutzt. Weiterhin werden für den Aufbau der Verbindung zur Datenbank und der Verarbeitung der Daten spezielle von *Oracle* bereitgestellte Javapakete verwendet, um eine gute Performance und Kompatibilität zu erreichen.
Zur Nutzung der Textsuche mit einem *CCMS* wurden diese in ein Modul eingebettet. Dafür wird durch die Klasse *OracleTextModule* eine Modulschnittstelle bereitgestellt. Durch die den Zusammenschluss mit dem *client module* zu einer Komponente wird die Abbildung auf das auf *Assets* gestützte System erreicht.
Ferner sind zusätzliche Klassen zum Zwecke der Verbindung und der Angleichung der beiden Ebenen nötig. Darüber hinaus wurden weitere Klassen erstellt, die administrative Aufgaben erledigen oder für eine zusätzlichen Bearbeitung der Daten zuständig sind.

4.2 Abstrakte Oberklasse

Kernstück des Moduls ist die abstrakte Klasse *OracleTextModule*. Sie implementiert die *mapping module* Schnittstelle und damit die Basisoperationen zum Verwalten der Assets und zur Steuerung des Moduls (siehe Abbildung 4.1). Sie ist *abstrakt* deklariert, damit die Umsetzung aller Methoden, die besondere Aspekte eines Assetmodells betreffen, von konkreten, schemaspezifischen Subklassen übernommen werden kann. Alle Modulrepräsentierungen sind also vom Typ *OracleTextModule*. Die Erläuterungen dieses Zusammenhangs und der Umsetzung sind Thema der folgenden Abschnitte.

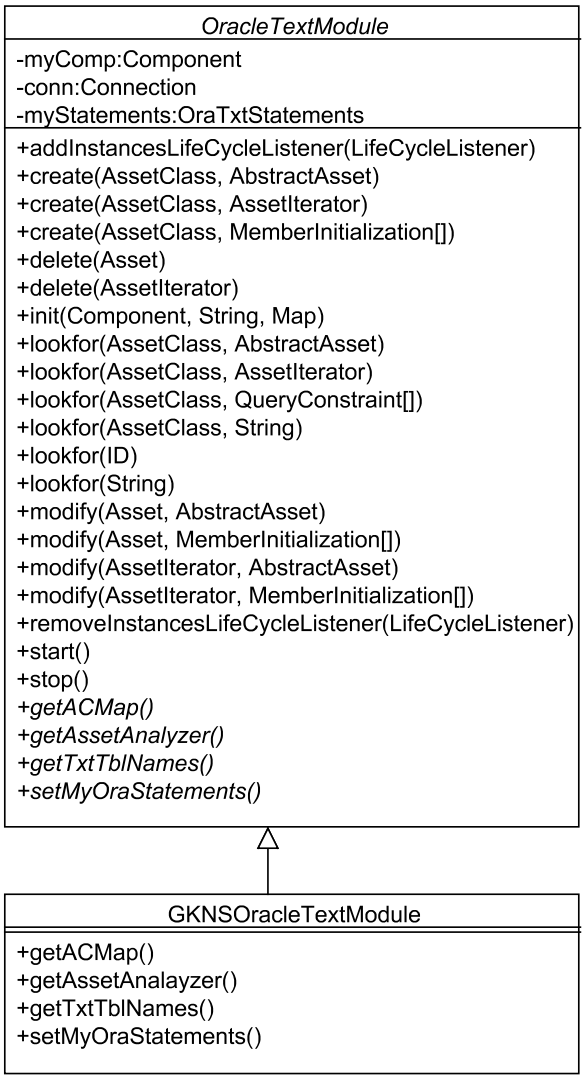

Abbildung 4.1: Abstrakte Klasse *OracleTextModule* unter Berücksichtigung der Vererbung

4.2.1 Konkrete modellspezifische Unterklasse

Die Verwirklichung des *OracleTextModule* für das GKNS-Schema geschieht durch die Klasse ***GKNSOracleTextModule***. Dementsprechend realisiert sie die abstrakten Methoden und überschreibt diese mit einer Implementation, die durch Benutzung der Informationen aus dem Assetdefinitionen den Umgang mit dem GKNS-Modell ermöglicht. Bedeutsam für diese Vorgehensweise ist die Methode `getTxtTblNames()`. Diese ist in gewisser Weise für die Initialisierung des *GKNSOracleTextModule* Moduls zuständig. Das bedeutet, ihr Hauptzweck besteht in der Bereitstellung der Informationen des Assetmodell für alle Modulklassen - und das vollführt sie auf 2 Arten. Das erste Vorgehen beinhaltet das Liefern einer Liste mit Objekten der Klasse `TableWrapper`. TableWrapper sind Objekte, die alle für das Programm nötigen Daten von Tabellen enthalten. Das sind, neben den Tabellennamen der Name des Indexes für die Tabelle, der Name des *Datastore-Objektes* und der Name der Spalte auf den der Index zeigt.

Die andere Art entspricht einer gleichzeitigen Erstellung von `AssetClassWrappern` und deren Ablegen in einer `HashMap`. `AssetClassWrapper` erfüllen den gleichen Zweck wie `TableWrapper`, nur auf einer logisch abstrakteren Ebene. Diese Objekte fassen alle Daten rund um die Assetklasse zentral zusammen und ermöglichen durch deren Zusammenlegung in einer Objektsammlung einen Zugriff auf das gesamte Assetmodell. Dabei wird durch sie nicht nur der Assetklassennamen bereitgestellt, sondern unter anderem auch eine Liste mit allen Tabellen, die zur Klasse gehören, wie einer Liste mit weitern Assetklassen die strukturell auf dieser aufbauen.
Alle Informationen über das Schema bezieht `getTxtTblNames()` aus extra Schema-Dateien, die von einer zusätzlichen Methode ausgelesen und in Gestalt eines `Properties` Objektes verfügbar gemacht werden. Das hat den Vorteil gegenüber der Implementierung des Modells in den Programmcode, dass Änderungen an den Assetdefinitionen auch ohne Neukompilierung des Quelltextes angepasst werden können.

In Beziehung dazu steht die Konkretisierung der Methode `getACMap()`. Prinzipiell handelt es sich dabei um eine simple Abfragemethode (*Getter*). Sie überprüft, ob eine Instanz der Hashmap für die Assetklasse existiert, erstellt gegebenenfalls eine und liefert diese als Rückgabewert wieder. Das Besondere ist, dass das Assetmodell durch diese Umsetzung, in Form der Sammlung aller Assetklassenwrapper, für das OracleTextModules verfügbar ist.

Ebenfalls eine *Getter-Methode* ist `getAssetAnalyzer()`. Genauso wie bei `getACMap()` wird das Objekt des Typs `AssetAnalyzer` aus der GKNS-Klasse abgefragt. Falls es noch nicht vorhanden ist, wird eine neue Instanz erstellt.
Einen besonderen Part übernimmt allerdings die Klasse `AssetAnalyzer.` Sie widerspricht in gewisser Weise dem Versuch der Unabhängigkeit vom Assetmodell, denn die Klasse beinhaltet weite Teile des GKNS-Assetschemas in ihrem Programmcode.
Daraus wird allerdings der Vorteil gezogen, dass Typumwandlungen, durch entsprechende *Cast*-Ausdrücke, genereller Asset Objekte in spezielle Typen des GKNS Schemas möglich sind. Dieser Vorgang ist für die Arbeitsweise des Klasse substanziell.

Der Zweck eines *AssetAnalyzers* besteht schließlich darin, ein gegebenes (unbekanntes) Asset Objekt zu analysieren und einer schemaeigenen Klasse zuzuordnen, um so auf seine speziellen Attribute Zugriff zu bekommen. Damit ist es ihm möglich, durch den Aufruf seiner Methode `getQueryString()` eine Anfrage zu liefern, die aus allen Attributwerten eines Assets generiert ist und für die Ähnlichkeitssuche nach gleichartigen Assets verwendet wird.
Der *AssetAnalyzer* entspricht dabei allerdings nicht dem Entwurfsmuster eines *Singletons* (GHJV95), denn einige Assetattribute haben einen Datentyp, der die Schnittstelle `Iterator` realisiert und daher auf eine Menge von anderen Assets verweisen kann. In diesem Fall wird für jedes Asset eine eigene Instanz des *AssetAnalyzers* erstellt.

Mit `setMyOraStatements()` gibt es außerdem eine *Setter-Methode*. In dieser Umsetzung werden aber nicht nur Objekte gesetzt, sondern zusätzlich noch Operationen ausgeführt.
Diese Operationen stehen nur indirekt mit den Assetmodell in Beziehung. Sie resultieren nicht aus den Assetdefinitionen, sondern sind für Modellspezifische Einstellungen der Suchmaschine verantwortlich. Dafür wird zuerst durch den Aufruf der *Setter-Methode* `setMyStatements()` der Super-Klasse eine neue Instanz des `OraTxtStatements` Objektes an die Super-Klasse übergeben.
Danach folgt dann mittels der Instanz eine Reihe an Methodenaufrufen, die die Objekte des Indizierungsprozesses für den Betrieb der Informationsrückgewinnung initialisieren. Zum Schluss wird der Aufruf zur Erschließung aller noch nicht eingebundenen Inhalte initiiert. Beim Erstbetrieb bedeutet das eine komplette Neuindizierung aller Text-Daten in der Datenbank, was unter Umständen einen hohen Zeitaufwand beanspruchen kann.
Aufgrund dieser Beschaffenheit der Konkretisierung der `setMyOraStatements()` Methode spielt sie beim Initialisierungsprozess des Moduls eine wichtige Rolle.

4.2.2 Implementierung der Modulschnittstelle

Von zentraler Bedeutung für die Funktionsweise sind die Methoden zur Steuerung des Moduls. Durch `init()`, `start()`, und `stop()` werden essenzielle Informationen eingeholt und wichtige Vor- und Nachleistungen erbracht.
`Init()` initialisiert das Modul, indem es beim Start der Komponente den Komponentennamen, den Modulnamen und alle Parameter aus der Konfigurationsdatei innerhalb des Moduls bekannt macht.
Die Methode `start()` baut danach die Verbindung zur Oracle Datenbank auf, unter Benutzung der Parameter, die von der init() Methode bezogen wurden. Dafür erstellt sie ein `OraConnections` Objekt, welches die Verbindungseigenschaften für die Oracle Datenbank bündelt. Dieses `OraConnections` Objekt bekommt die Parameter in Form eines `Properties` Objektes und liefert dann die „Verbindung“ zurück .
Darüber hinaus ruft sie die abstrakte Methode `setMyOraStatements()` auf. Diese wird durch die jeweilige modellspezifische Unterklasse realisiert und stellt den betriebsbe-

reiten Zustand der Suchmaschine für die Rückgewinnungsfunktionalität her.
Mit dem Aufruf der `stop()` Methode wird die Beendigung des Moduls eingeleitet. Bei dieser Gelegenheit findet eine Überprüfung statt, ob eine aktive Verbindung zur Datenbank besteht. Ist dies der Fall, wird die Sitzung geschlossen.

Die Implementierung der Operationen zum Managen der Assets aus der einheitlichen Schnittstelle im *OracleTextModule* entspricht in den meisten Fällen einer Weiterdelegierung der Aufgaben an das Basismodul. Dadurch übernimmt das Basismodul die Abbildung des Models auf die Datenbank bei der Ausführung der Operation.
Grundlegend für die Funktionsweise des *OracleTextModule* ist jedoch, dass jede durch die Durchführung dieser Operation vollzogenen Änderungen in der Datenbank auch im zugehörigen Index aktualisiert wird. Dafür wird bei jeder Ausführung mit Hilfe der *AssetClassMap*, die vertikale Struktur der betreffenden Assets und zugleich die in den Synchronisierungsprozess einzuschließenden Tabellen ermittelt. Nach der Abwicklung der eigentlichen Operation in der Datenbank wird dann für jede Tabelle eine `sync(tablename)` Methode ausgeführt, die den Index auf den neusten Stand bringt.

Listing 4.1 zeigt anhand des Beispiels der Implementierung der Methode zur Erstellung eines einzigen *Assets*, wie die Weiterreichung an das Basismodul (rot hervorgehoben) und die anschließende Synchronisation der entsprechenden Indizes umgesetzt ist.

Listing 4.1: Erstellung eines *Assets*

```
1  public Asset create(AssetClass cClass,
2                      AbstractAsset abstractCreateAsset) {
3
4  ->      Module base = myComp.getBaseModule(this);  <-
5  ->      Asset a = base.create(cClass, abstractCreateAsset);  <-
6
7          AssetClassWrapper aCWrap = getACMap().get(cClass.getName());
8          aCWrap.setACMap(getACMap());
9          ArrayList<String> tableNames = aCWrap.getFullTblList();
10
11         Iterator<String> tIter = tableNames.iterator();
12         while (tIter.hasNext()) {
13                 String tabName = tIter.next();
14                 myStatements.syncAll(tabName);
15         }
16                 return a;
17 } // create
```

Dieser Beispielquelltext verweigert sich dem Anspruch auf Korrektheit gegenüber der Implementierung.

Wiedergewinnungsmethoden

Anders wird es in den Realisierungen der `lookfor()` Operation geregelt. Diese benutzen im ersten Schritt die unterliegenden Strukturen der *Oracle Text* Suchmaschine um eigenständig die Anfragen zu beantworten. Erst danach wird die Leistung des Basismoduls genutzt, um die aufgefunden Informationen für ein *CCMS* nutzbar zu machen. Beim Durchsuchen des kompletten Datenbestands nach einem beliebigen Anfrageterm, wird innerhalb der Schnittstellenimplementierung, die Methode `lookfor(queryString)` einer *OraTxtStatements* Instanz ausgeführt. Diese liefert eine Liste des Typs *ArrayList* zurück, in der die Identifier (cocoma_ids) der erhaltenen Resultate gespeichert sind.

```
ArrayList<Integer> idList;
idList = myStatements.lookfor(queryString);
```

Diese Primärschlüssel der Datenbank werden nun durch die Benutzung der `lookfor(ID)` Methode, die durch die Weiterreichung der Anfrage an das Basismodul umgesetzt ist, auf ihre betreffenden Assets abgebildet. Zuständig dafür ist eine Unterklasse vom *AssetIterator* der ***OraTextAssetIterator***. Beim zugehörigen Aufruf von `nextAsset()` wird zur aktuellen *cocoma_id* das jeweilige Asset geliefert.
Der *OraTextAssetIterator* wird dann als Rückgabewert stellvertretend für einen *AssetIterator* zurückgegeben.

```
OraTextAssetIterator oraAI = new OraTextAssetIterator(
myComp, baseModuleName, idList);
return oraAI;
```

Kommt in der Methodensignatur noch ein Parameter des Typs *AssetClass* als Eingrenzung des Suchraumes hinzu, werden ausschließlich Assets der festgelegten Assetklasse inklusiver aller Unterklassen in die Suche eingebunden. Um die dafür einzuschließenden Tabellen zu ermitteln, wird sich einer *HashMap* bedient, die eine Sammlung von AssetClassWrapper enthält, welche es ermöglichen die Struktur des Schemamodels zu erfassen.

```
AssetClassWrapper aClassWrap = getACMap().get(qryClass.getName());
aClassWrap.setACMap(getACMap());
ArrayList<String> tableNames = aClassWrap.getFullTblList();
```

Aus der *aCMap* Hashmap wird der richtige AssetClassWrapper des jeweiligen Assets bezogen. Dieser enthält alle notwendigen Informationen über die Assetklasse und erlaubt es damit, eine komplette Liste der Tabellennamen zu liefern, die für die Suche relevant sind.

Für die Ähnlichkeitssuche zwischen Assets werden ein oder mehrere Beispielassets in die Parameterliste aufgenommen. Anhand dieser wird dann in den erschlossenen Assetinformationen nach homologen Assetobjekten recherchiert. Dafür ist es notwendig, die

erhaltenen Assets erstmals bezüglich ihrer Struktur zu untersuchen, um sie daraufhin aufgliedern zu können. Mittels der durch das Aufgliedern erhaltenen Attributwerte der Assets kann dann eine entsprechende Anfrage formuliert werden, die die gewünschten Resultate liefert. Verwirklicht wird die Analyse und das Erstellen eines Anfrageterms durch ein *AssetAnalyzer* Objekt.
Ab diesem Teilschritt unterscheidet sich die Implementation nicht mehr von einer auf eine Assetklasse eingegrenzten `lookfor()` Realisierung und kann darauf zurückgeführt werden.

```
String queryString = "(";
getAssetAnalyzer().prepareQuery(example);
queryString += getAssetAnalyzer().getQueryString();
queryString += ")";
return this.lookfor(queriedClass, queryString);
```

4.3 Verwirklichung der inhaltlichen Rückgewinnungsstruktur

Die Umsetzung der Klassen zur datennahen Rückgewinnung unterscheidet sich an einigen Stellen von der des Entwurfes. Für das essenzielle Element, die Erschließung der Daten, stellte sich heraus, dass einige Objekte des Indizierungsprozesses aus Anforderungssicht keiner besonderen Konfiguration bedürfen. Als Resultat entstanden demnach lediglich 3 der eigentlich 6 vorgesehenen Klassen, die jeweils ein Datenbankobjekt abbilden. Direkt mit der Erschaffung der zentralen *OraTxtStatements* Instanz werden, die mit ihr eine Komposition bildenden, *OraTxtWordlist* und *OraTxtLexer* Objekte instanziiert. Durch sie ist eine Konsistenzkontrolle sowie die richtige Konfiguration der erzeugten Datenbankobjekte gewährleistet.
Darüber hinaus ist das Verwalten des Datenbank-Objekts *DATASTORE* durch die *OraTxtDatastore* Klasse sichergestellt. Bei jeder Indexerstellung muss ein *DATASTORE* Objekt angegeben werden, weil es festlegt, woher die Daten zu beziehen sind und in speziellen Fällen auch, welche Daten überhaupt erschlossen werden sollen. Da jede zu indizierende Tabelle lediglich einen Index bekommt, aber alle infrage kommenden Textspalten eingeschlossen werden sollen, sind eine entsprechende Menge an zu erstellenden *MULTI_COLUMN_DATASTORE* Präferenzen notwendig. Dies wird mittels der Methode `create()` der jeweiligen *OraTxtDatastore* Instanz geleistet. Dabei muss eine Unterscheidung zwischen den Spalten, die reine textbasierte Typen aufweisen, und denen, die erst durch Benutzung der SQL Funktion `TO_CHAR()` lesbar gemacht werden müssen, vorgenommen werden. Der zweite Fall betrifft vornehmlich Spalten mit numerischen Typen, wie z. B. Zeitstempel.

Erweiterungen des Entwurfes ergaben sich durch die praktische Ausführung klassischer Suchmaschinen Dienste. So kamen zwei Klassen hinzu, die verarbeitenden und analytischen Charakter aufweisen und als Werkzeuge für die Arbeit mit *Oracle Text* dienen.

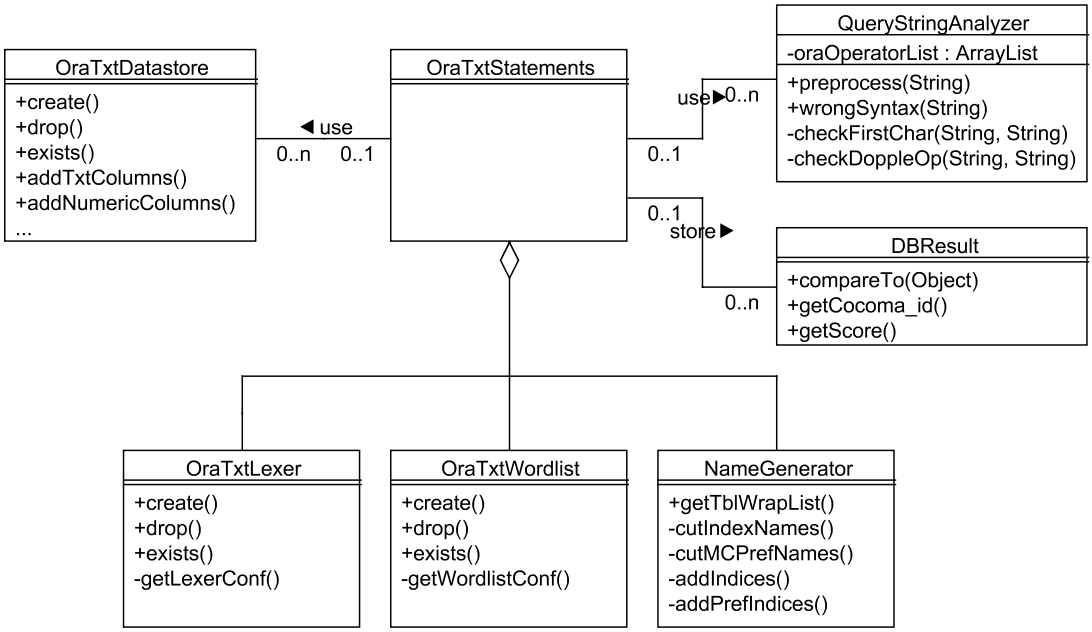

Abbildung 4.2: Aufbau der Klassen für Oracle Text

Unterstützende Klassen

Bei der Erstellung von Datenbankobjektes ist aufgrund der Beschränkung der Länge der Bezeichner eine Klasse *NameGenerator* erforderlich. Speziell die Namen der Indizes und der *Datastore* Objekte sind anzupassen. So dürfen Bezeichner eines Indizes nicht über mehr als 25 und die einer *MULTI_COLUMN_DATASTORE* Präferenz nicht über mehr als 30 Zeichen verfügen. Der *NameGenerator* generiert demnach, auf Basis des zugehörigen Tabellennamens, die jeweils ihre Spezifikation entsprechenden Namen, der in der Datenbank für das betreffende Schema stehenden Elemente. Dabei sind Indizes mit einem „IDX_“ und *MULTI_COLUMN_DATASTORE* Präferenzen mit einem „MCP_“ Präfix und zusätzlich einer durchlaufenden Nummerierung an ihrem Ende gekennzeichnet.

Der *QueryStringAnalyzer* übernimmt gleich zwei Aufgaben. Einmal dient er als Kontrollinstanz vor der Ausführung einer Suchanfrage um Datenbank Fehler durch falsche Anfrageterme zu vermeiden und überdies nimmt er eine Vorverarbeitung der Anfrageterme der Ähnlichkeitssuche vor, damit diese nicht durch ungewollte Ausdrücke verfälschte Resultate erzeugen.
Vor jeder Ausführung einer Suche wird zuerst der sogenannte *QueryString* analysiert. Dabei wird darauf geachtet, das keine inkompatible Syntax der Anfrageoperatoren vorliegt. *Oracle Text* erlaubt es zum Beispiel nicht, dass Operatoren den Ausdrücken vorgestellt sind oder dass zwei Operatoren (auch unterschiedliche) direkt aufeinander folgen ohne dass ein Operand zwischen ihnen steht. Für diese Aufgabe enthält die Klasse eine Liste mit allen Operatoren, die in *Oracle Text* verwendet werden.

Die Vorverarbeitung geschieht mit der für die Suche nach analogen Instanzen, aus den Attributwerten der zu berücksichtigen Assets, generierten Anfrage. Sie ist deshalb notwendig, weil einige Assets über lange Passagen freien Textes als Inhalte verfügen. Dieser fließt dann als Literal in die *Konjunktive Normalform* der Anfrage ein. Die Textpassagen in der Formel sind deshalb ein Problem, weil sie etliche syntaktische Zeichen zur Strukturierung und Verständlichkeit ihrer Phrasen und Sätze enthalten, die aber gleichermaßen Schlüsselwörter für die Suchmaschine sind! So wie hier gerade das Ausrufezeichen („**!**“) zur Bekräftigung der Aussage und einige Kommata zur Trennung ihrer Elemente („**,**“) verwendet wurden, tritt dies bei den meisten dokumentartigen Texten auf - ihre Verwendung ist sogar unverzichtbar. Allerdings entspricht ein „**,**“ in einer Anfrage dem `ACCUMulate` Operator, der die Häufigkeit des Auftretens der Terme in den Dokumenten als Trefferzahl wiedergibt. Das „**!**“ dagegen ist ein Synonym für den `soundex` Operator. Dieser bewirkt, dass zusätzlich Treffer angezeigt werden, die ähnlich „klingen“ wie der Angegebene. Diese und einige mehr der üblichen Satzzeichen werden durch den *QueryStringAnalyzer* entfernt und dadurch wird eine ungewollte Beeinflussung der Suchergebnisse vermieden.

4.4 Probleme in der Umsetzung

Während der Umsetzung des Designs in eine lauffähige Komponente eines *CCM* Systems gab es ein paar Punkte, die eine intensivere Handhabung erforderten. Im Folgenden werde ich einige dieser Schwierigkeiten diskutieren. Dazu sind sie in zwei Kategorien aufgeteilt. Zuerst werden die besprochen, die sich bei der Integration in ein konzeptionelles Inhaltsverwaltungssystem ergaben. Danach widme ich mich Problemen, die während der Arbeit mit der externen Suchmaschine von Oracle auftraten.

4.4.1 Probleme mit einem CCMS

Während der Realisierung trat vor allem immer wieder eine Schwierigkeit auf, das Fehlen der Informationen des zugehörigen Assetmodells. Aufgrund der Tatsache, dass das *OracleTextModule* Modul nicht wie andere Module vom Modelcompiler generiert wird, müssen die Angaben, die sonst durch die Assetdefinitionen geliefert werden, immer eigens erarbeitet werden. Bereits beim Grobdesign ergab sich daraus eine Aufspaltung der Realisierung in schemaspezifische und schemaunabhängige Elemente in eine hierarchische Klassenstruktur.

Vornehmlich sind ständig Informationen über die Assetklassen und deren strukturelle Beziehungen bei der Handhabung der Assets notwendig. So musste, neben der Beschaffung und Speicherung der betreffenden Daten auch ein effektives Management dieser aufgebaut werden, da ihr Einsatz in weiten Teilen des Moduls erforderlich ist. Daraus entstand die Klasse *AssetClassWrapper* und ihre dazugehörige Sammlung, die *ACMap*.

Ebenfalls das gleiche Problem ergab sich auf Daten-Ebene. Hier fehlten vorweg die Kenntnisse über das Datenbankschema, welches ebenfalls generiert wird. Dafür ist der *SQL-Schemagenerator* zuständig, der nach einem bestimmten Muster die Datenbank mit den Tabellen für die Assetwerte aufbaut. Abgesehen von der Regelung des Zugriffs auf die Tabelleninformationen innerhalb der Software, besteht in diesem Zusammenhang primär die Schwierigkeit darin, die Brücke zu schlagen zwischen den Assets und ihren assoziierten Tabellen. Zum Verwalten des Datenbankschemas sind die Objekte der Klasse *TableWrapper* und ihre Kollektion des Typs *Hashmap* eingerichtet worden. Besonders wertvoll sind die *TableWrapper* vor allem deswegen, weil sie eine zur Laufzeit persistente Speicherung der Benennungen der sich auf die Tabellen beziehenden Datenbankpräferenzen erlauben.

4.4.2 Probleme mit der Suchmaschine

Bei der Zusammenarbeit mit der Suchmaschine *Oracle Text* traten während der Implementierung einige Schwierigkeiten auf.
Einen Punkt stellte dabei generell die Fehlerbehandlung von *Oracle Text* dar. So werden auftretende Fehler beim Indizierungsprozess zwar mit der Beendigung desgleichen und einer erläuternden Meldung gehandhabt, aber dennoch wird ein Index-Objekt für die entsprechende Tabelle angelegt. Die Problematik entsteht dann, wenn bei Verwendung Folgefehler aufgrund der Unbenutzbarkeit entstehen, zugleich aber keine neuen Indizes erzeugt werden können, weil bereits einer besteht. Lösen kann man den Zustand nur durch Erkennung und Löschung des Index-Objektes. Demnach resultierten daraus Fehlerbehandlungsroutinen, deren Aufgabe im Löschen und Neuanlegen von (potenziell) fehlerhaften Indizes besteht und die immer dann gestartet werden, wenn es zu irgendeiner Meldung vonseiten der Suchmaschine kommt.

Schwierigkeiten ergaben sich auch durch den Ablauf des Indizierungsprozesses im Zusammenhang mit der unzureichenden Unterstützung von Spalten, deren Einträge nicht auf einem Texttyp basieren. Für die Erstellung eines Indizes benötigt nämlich das *DATASTORE* Element zwingend eine Tabelle mit mindestens einer Textspalte. Zwar ist es möglich, auch andere Spalten als nur welche mit Textformate in einen Index aufzunehmen, indem man ihre Werte mit Hilfe einer Funktion konvertiert, aber die Schaffung eines Indizes basierend auf eben so einer Spalte ist nicht zulässig. Dadurch fallen die Tabellen heraus, die lediglich Spalten ohne Textformate aufweisen, aber evtl. trotzdem relevante Informationen anderen Typs enthalten. Außerdem gibt es Probleme bei der Ausführung einer Ähnlichkeitssuche mit mehreren Beispielassets, wenn die generierte Anfrage durch zu lange oder zu viele Attributwerte eine Länge von 4000 Zeichen überschreitet. Tritt dieser Fall ein, kommt es zu einer Datenbank-Fehlermeldung und dem Abbruch der Anfrage. Zur Zeit ist dies nur durch eine Warnung der Software gehandhabt und nicht gelöst.

5 Zusammenfassung und Ausblick

Am Schluss dieser Arbeit werden die gewonnenen Ergebnisse zusammengetragen. Des Weiteren wird aufbauend auf den zuvor erarbeiteten Grundlagen ein Ausblick auf weiterführende Tätigkeiten gegeben, die sinnvollerweise in Zukunft vorzunehmen sind.

5.1 Zusammenfassung der Arbeit

Als Resultat dieser Projektarbeit liegt das in dieser Arbeit entworfene und in Java implementierte Modul **OracleTextModule** vor.
Eingebettet in eine Komponente dient es als eine für das *GKNS*-Projekt entwickelte Erweiterung eines *konzeptionellen Inhaltsverwaltungssystems* (*CCMSs*).
Das Modul ist ein prototypischer Entwurf für die Erstellung eines Generators. Die hierdurch gewonnenen Erkenntnisse dieser Arbeit liegen ausschließlich im Bereich der Einbindung und Umsetzung einer Volltextsuche und nicht in der Generatorerstellung. Folgende Anwendungsfelder werden durch das Modul abgedeckt:

- Erschließung der in den Entitäten enthaltenen Informationen
- Rückgewinnung und Bereitstellung der Informationen für ein *CCMS*

Zur Erschließung und Wiedergewinnung der Daten wird die Technologie der Fremdsoftware *Oracle Text* verwendet. Die Abbildung der erhaltenen Resultate auf das *Assetmodell* wird durch die Implementierung einer Modulschnittstelle innerhalb des Moduls geleistet.

Die Grundlage für den Generator sind die Erkenntnisse in Bezug auf die Funktions- und Arbeitsweise von *Oracle Text* und der Methodik ihrer Eingliederung in eine Instanz des *CCM*.
Zur Erarbeitung dieser, findet in Kapitel 2 eine Einarbeitung in die Konzepte des *Conceptual Content Management* statt. Diese stellt den Aufbau der zweiteiligen Assets und ihren Beziehungen zueinander dar und liefert eine Übersicht über die aus Komponenten und Modulen bestehende und auf Basis von Assetdefinition generierte Architektur des Systems. Darüber hinaus wird im weiteren Verlauf des 2. Kapitels die Funktionsweise und Bedienbarkeit von *Oracle Text* erläutert. Hier geht es vor allem, um den aus fünf Elementen bestehenden Indizierungsprozess und um die Abfragesyntax für die vier vorhandenen Indextypen. Dabei zeigt sich die Verwendung eines *CONTEXT*-Index in Verbindung mit seinem Abfrageoperator *CONTAINS* für die Suche in größeren Textdokumenten am geeignetsten. Die Analyse zeigt auch, dass Oracle Text

den Anforderungen speziell in dem Bereich der assoziativen Suche entspricht. Die ausschlaggebenden Vorteile liegen im Vergleich zu vorhandenen alternativen Technologien vorwiegend in der Suchqualität der Ergebnisse.

Das in Kapitel 3 vorgestellte Design zeigt, dass die Integration durch eine aus zwei Modulen bestehende Komponente zu realisieren ist. Die Umsetzung der Erschließung sowie der anschließenden Textsuche leistet das *OracleTextModule* Modul, während das Basis Modul der Komponente für die Abbildung der Resultate auf die Assettechnologie sorgt. Grundlegend ist dabei die Trennung der Beschaffung und Verarbeitung von Daten auf der einen Seite und die Einbindung in ein *CCM* durch die Implementierung einer Modulschnittstelle innerhalb des *OracleTextModule* Moduls auf der anderen Seite.

Durch die Implementierung (Kapitel 5) hat sich herausgestellt, dass die Aufgliederung der Struktur in eine abstrakte und für jedes Schema eine konkretisierende Klasse ohne Generator notwendig ist, um nicht völlig auf die Generalität des Moduls verzichten zu müssen. Außerdem zeigte sich die Notwendigkeit von Strukturen, die besondere Ereignisse der Datenbank behandeln. Dazu gehören eine Analyse der Anfragen, um Fehler zu vermeiden, genauso wie eine Generierung von allen verwendeten Bezeichnern aufgrund von datenbankinternen Beschränkungen. Entscheidend bei der Komponente ist auch die Erzeugung und Verwendung von aus mehreren ausgewählten Spalten bestehenden Indizes. Dies filtert nicht nur ungewollte Spalten heraus, sondern bewirkt auch eine sich aus Anforderungssicht ergebene Performancesteigerung.
Nicht zu vernachlässigen für das Ergebnis sind auch die Probleme, die während der Arbeit entstanden. Dies betrifft vorwiegend die Extrabehandlung von Fehlern, die während der Indizierung auftauchen und die fehlende Unterstützung nichttextbasierter Datentypen für die Textsuche.

5.2 Ausblick in Bezug auf das Thema

Durch die Realisierung einer auf *Oracle Text* basierenden Komponente zur Textsuche mit einem *CCMS* ist die Grundlage für Folgearbeiten im Bereich der professionellen Suche geschaffen. Diese weiterführenden Tätigkeiten sehe ich aufgegliedert in zwei Kernabschnitte. Der eine betrifft den Bereich der weiteren und tiefer gehenden Integration in die Architektur eines *konzeptionellen Inhaltsverwaltungssystems*. Der andere entspricht einem Ausbau der Möglichkeiten, die man erhält durch Benutzung einer so komplexen und umfangreichen Suchmaschine, wie die hier verwendete.

5.2.1 Anfertigung eines Generators

Einer der größten Nachteile des Moduls ist sicherlich die relative Abhängigkeit vom eingearbeiteten und bereitgestellten Assetschema. Ebenso entsprach es einer der größten Aufgaben während der Erstellung des Moduls, die fehlenden Informationen aus den

Assetdefinitionen zu erhalten und zu verwalten.
Deswegen ist der Bau eines Generators, der eine Komponente erstellt, die diese Funktionalität enthält, eine der primären Ansätze zur Verbesserung der Software.
Ein aus einem Generator erstelltes Modul für die Suche mit *Oracle Text* in einem *CCM* würde demnach um einige Teile der Umsetzung schlanker und würde zugleich ein effizienteres Arbeiten ermöglichen. Außerdem könnte es auf die schemabeschreibenden Dateien verzichten und würde keinen Verwaltungsaufwand erfordern. Bedeutsam ist vor allem, dass es die Konzepte der *Offenheit* und der *Dynamik* des *CCM* komplett umsetzen würde.

5.2.2 Funktionserweiterungen

Eine andere Aufgabe wäre die weitere Ausnutzung der Möglichkeiten der Suchmaschine für ein *konzeptionelles Inhaltsverwaltungssystem*. Das betrifft zum einem die jetzige Realisierung der Methoden zur Suche aus der Schnittstelle für die Module und zum anderen den Ausbau genau dieser Schnittstelle durch zusätzliche Operationen. Soll heißen, durch die *Oracle Text* Technologie werden unzählige Operatoren und Funktionen bereitgestellt, die alle spezielle Suchmethoden umsetzen. Speziell im Bereich der Assoziativen und der Äquivalenzsuche gibt es weitere Ansätze (Rei06), bei denen zu erarbeiten wäre, inwieweit sie das aktuelle Konzept ausbauen können.
Da wären zum Beispiel Methoden um Synonyme (SYN), bevorzugte Begriffe (BT), verwandte Begriffe (RT), Oberbegriffe (BT) und Unterbegriffe (NT) zu ermitteln, um nur eine Auswahl zu geben. Damit könnte man Beispielweise den Suchraum für „Java“ um folgende Begriffe weiter aufspannen:
- Programmiersprache, Kaffeebohne, Insel *(BT)*
- Beans, Swing, Servlets,... *(RT)*

und viele weitere.
Ebenso sollte diskutiert werden, inwieweit die Benutzung des Operators *(TR)*, der alle äquivalenten Begriffe anderer definierter Sprachen mit einbindet, sinnvoll ist.

Zusätzliche Einsatzmöglichkeiten für das Modul könnten sich durch den Gebrauch von zielorientierten Suchoperationen ergeben. So bietet sich eine Suche an, die eingeschränkt auf nur einem oder einer definierten Menge aller vorhanden Attribute eines Assets durchgeführt wird. Es ist sogar möglich, mithilfe des *within* Operators die Suche so einzuschränken, dass nur Sektionen, Absätze oder Sätze eines einzigen Attributs durchsucht werden.

Darüber hinaus sollte erörtert werden, inwieweit auch Indexe anderen Typs, entweder anstelle des Bestehenden oder zusätzlich zu ihm, einen Vorteil bringen könnten. Der verwendete Indextyp *CONTEXT* ist im Prinzip ausgelegt für Volltext-Suchanfragen, die auch große Textdokumente einschließen. Sind bedingt durch das Schema allerdings viele Typen unterschiedlichen Formats in den Erschließungsprozess mit einzuschließen, bietet sich der *CTXCAT* Typ für den Index an. Durch Erstellung von Indizes des Typs *CTXRULE* sind sogar selbst definierte oder eigenständig erstellte Klassifikationen der Informationsobjekte möglich. Diese Möglichkeit hat allerdings den Nachteil, dass die

Umsetzung nicht ohne Erstellung weiterer Tabellen in der Datenbank auskommt.
Durch die Verbindung der beiden Technologien von *Oracle Text* als Suchmaschine und einem *CCMS* sind eine Vielzahl an potenziellen Weiterentwicklungsmöglichkeiten entstanden.

6 Literaturverzeichnis

BYa Baeza-Yates, Ricardo ; al. et: *Indexing Methods for Approximate Text Retrieval (Extended Abstract)*. citeseer.ist.psu.edu/160628.html 2.2.4

Fer03 Ferber, Reginald: *Information Retrieval*. Heidelberg : dpunkt.verlag, 2003 1.1

GHJV95 Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2 4.2.1

Hei06 Heidenreich, Andreas ; StudyPaper.com (Hrsg.): *Lucene - Eine einfache Suchmaschine für Freitextsuche*. Version: 2006. http://www.lucene.de/, Abruf: 07.09.2006. DomainLoc.com GmbH 2.4

Lan02 Langer, Stefan: Grenzen der Sprachenidentifizierung, CIS, Universität München, 2002 (Tagungsband KONVENS, Saarbrücken), S. 99–106 2.2.2

Lee94 Lee, Joon H.: Properties of extended Boolean models in information retrieval. In: *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*. New York, NY, USA : Springer-Verlag New York, Inc., 1994. – ISBN 0–387–19889–X, S. 182–190 2.2.2

MS98 Matthes, Florian ; Schmidt, J.W.: *Datenbankhandbuch*. Technische Universität Hamburg-Harburg, 1998 2.2.4

OMG06 *Object Management Group - UML*. Version: 2006. http://www.uml.org/#UML2.0, Abruf: 07.09.2006. Object Management Group 1.4

OT006 *Oracle Technology Network - Oracle Text*. Version: 2006. http://www.oracle.com/technology/products/text/index.html, Abruf: 19.09. 2006. Oracle Corporation 2.3, 2.4

OTF05 *Oracle Text - Feature Overview*. Oracle Parkway, Redwood Shores, USA : Oracle Corporation, April 2005 2.3

OTR04 *Oracle Text - Reference 10g Realease 1*. Oracle Parkway, Redwood Shores, USA, June 2004 2.3.2

Pan86 Panyr, Jiri: Theorie der Fuzzy-Mengen und Information-Retrieval-Systeme, VCH Verlagsgesellschaft, 1986 (Nachrichten fuer Dokumentation 37), S. 163 – 168 2.2.2

Rei06 Reimer, Jörg: *Oracle Thesauri - mehr Nutzen in der Volltextsuche*. Hamburg, Februar 2006 5.2.2

Sch06 Schumann, Georg: Lucene - Volltextsuche mit Java / Fachhochschule Augsburg. 2006. – Forschungsbericht 2.4, 2.5

Seh04 Sehring, Hans-Werner: *Konzeptorientierte Inhaltsverwaltung Modell, Systemarchitektur und Prototypen*, Technische Universität Hamburg-Harburg, Diss., 2004 1.1, 2.1

Seh06 Sehring, Hans-Werner: *Open Dynamic Conceptual Content Management*. Version: 2006. http://www.sts.tu-harburg.de/~hw.sehring/cocoma/, Abruf: 07.09.2006. Softwaresysteme - Technische Universität Hamburg-Harburg 1.1

SMW06 Skulschus, Marco ; Michaelis, Samuel ; Wiederstein, Marcus: *Oracle 10g : das Programmierhandbuch*. 2006 3.4.2

SS03 Schmidt, Joachim W. ; Sehring, Hans-Werner: Conceptual Content Modeling and Management, The Rationale of an Asset Language. In: Alexandre V. Zamulin, Manfred B. (Hrsg.): *Proc. Perspectives of Systems Informatics (PSI'03)*, Springer-Verlag, 2003 (Lecture Notes in Computer Science 2890), S. 469–493 2.1.1

SS04 Sehring, Hans-Werner ; Schmidt, Joachim W.: Beyond Databases: An Asset Language for Conceptual Content Management. In: Gottlob, Georg (Hrsg.) ; Benczúr, András A. (Hrsg.) ; Demetrovics, János (Hrsg.): *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004*, Springer-Verlag, 2004 (Lecture Notes in Computer Science 3255), S. 99–112 2.1.2, 2.2

UML06 *UML 2.0 Tutorial*. Version: 2006. http://www.sparxsystems.com.au/resources/uml2_tutorial/index.html, Abruf: 07.09.2006. Sparx Systems Pty Ltd.