| | |
|---|---|
| **Studiengang:** | **Softwaretechnik** |
| **Betreuer:** | **Dipl.-Inform. Christoph Mangold** |
| | **Dr.-Ing. Marco Litto, mind8 GmbH** |
| **Prüfer:** | **Prof. Dr.-Ing. habil. Bernhard Mitschang** |
| **Beginn am:** | **28.02.2003** |
| **Beendet am:** | **28.08.2003** |
| **CR-Klassifikation:** | **H3.3, H3.1, H3.4** |

Diplomarbeit Nr. 2078

# Entwicklung eines Konzepts zur Volltextsuche in semantischen Netzen

Michael Wenig

Fakultät Elektrotechnik, Informatik,
Informationstechnik
Universität Stuttgart
**Institut für Parallele und Verteilte Systeme (IPVS)**
**Abteilung Anwendersoftware (AS)**
Universitätsstr. 38
70569 Stuttgart

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

## Kapitel 1

# Einleitung

Unter steigendem Kosten- und Wettbewerbsdruck werden gegenwärtig in vielen Bereichen massive Anstrengungen zur Verkürzung von Produktentwicklungszeiten bei gleichzeitiger Erhöhung der Qualität unternommen. Dies betrifft auch den Bereich des Maschinenbaus, und hier insbesondere den Sondermaschinenbau (Spezialmaschinen für einzelne Kunden).

Früher wurden oft Maschinen in Einzelfertigung erstellt, ohne dabei eine Wiederverwendung der Bestandteile bei anderen Maschinen zu berücksichtigen. Dies verhindert auch eine Wiederverwendung durch Kopieren, da die Anforderungen zwischen verschiedenen Maschinen selten identisch sind. Jede neue Entwicklung verursachte daher einen immensen Aufwand, da praktisch alle Teile von Grund auf neu entworfen, gefertigt und geprüft werden mussten. Dieses Verfahren wird mit wachsender Komplexität der Maschinen aufwändiger bis die Vielfalt nicht mehr beherrschbar ist.

In der Informatik reifte in den späten 60er Jahren die Erkenntnis, dass manche Software-Projekte selbst mit gigantischem Aufwand nicht zu einem zufriedenstellenden Ende geführt werden konnten. Dieses Problem wurde als *Software Crisis* bekannt und führte zur Entstehung des Software-Engineerings. Ziel des Software-Engineerings ist die Erstellung von Software in einem klaren Produktions-Prozess.

Eine Methode des Software-Engineerings ist die Modularisierung. Dabei werden die Systeme in beherrschbare Einheiten zerlegt und zunächst nur die Schnittstellen definiert. Jede Einheit wird dann einzeln entworfen, erstellt und getestet. Anschließend werden diese Einheiten zu einem Gesamtsystem zusammengesetzt. Eine offensichtliche Verbesserung der Prozesse und der Qualität besteht darin, diese Einheiten nicht nur in einem System, sondern in vielen Systemen zu verwenden. Dies vermindert die Kosten und erhöht die Qualität. Vorraussetzung ist jedoch, dass die Einheiten entsprechend universell entworfen werden. Sowohl in der Informatik als auch im Maschinenbau ist momentan ein starker Trend vorhanden, diesen Komponentenbau umzusetzen.

Ergebnis ist ein Baukastensystem mit vielen universellen Bausteinen. Neue Produkte werden durch Kombination und Konfiguration dieser Bausteine erstellt. Im Maschinenbau besteht ein solcher Baustein aus einer Mechanik, einer Elektrik, einer Steuerungssoftware und einer Dokumentation. Meist werden die Teilgebiete von unterschiedlichen Abteilungen betreut, wobei für jeden Bereich unterschiedliche Werkzeuge eingesetzt werden. Die mechanische und elektrische Konstruktion verwenden beispielsweise jeweils spezielle CAD-Systeme, die Software-Erstellung setzt entsprechende SPS-Entwicklungssysteme ein und die Dokumentation wird ebenfalls durch eine eigene Software unterstützt. Die Systeme bieten allerdings meist keine gemeinsamen Schnittstellen und verwenden ihre eigenen proprietären Daten-Ablage-Strukturen. Die Abhängigkeiten zwischen den Teilbereichen sind meist nur durch das spezielle Wissen der Mitarbeiter "dokumentiert", eine formale Definition existiert nicht.

Das Projekt *Foederal* ([12]) hat sich dem Problem der Abhängigkeiten zwischen den Bereichen angenommen und verknüpft die Informationen unter Verwendung semantischer Netze. Diese Netze enthalten alle vordefinierten Komponenten, sowie deren Parameter. Die erstellte *Föderale Informations Architektur* (***FIA***) bietet die Möglichkeit Maschinen aus diesen vordefinierten Komponenten durch Konfiguration zu modellieren. Aus dem Modell können dann die fertigen Steuerungsprogramme, Schaltpläne und Dokumentationen für die speziellen Maschinen generiert werden.

## 1.1 Inhalt der Arbeit

Die semantischen Netze enthalten viele Informationen, welche allerdings durch vorhandene Standard-Mechanismen nicht durchsucht werden können. Ziel dieser Arbeit ist daher die Konzeption einer Erweiterung um eine Volltextsuche. Dies erfolgt in Kooperation der Firma *mind8 GmbH*, welche die Architektur fachlich und technisch betreut.

Kapitel 1 beschreibt zunächst theoretische Grundlagen zur Problematik der Volltextsuche. Die Ausführungen beschränken sich dabei im Wesentlichen auf die in den weiteren Kapiteln verwendeten Konzepte und Methoden.

Da bereits die Ausschreibung dieser Diplomarbeit das OpenSource-Framework Apache-Lucene als geeignet erachtet, wurde keine Auswahl zwischen verschiedenen Produkten vorgenommen. In Kapitel 3 wird Lucene ausführlich untersucht und beschrieben. Die Ausführungen orientieren sich an den Arbeitsschritten von Lucene und werden jeweils von einem konzeptionellen und einem implementierungstechnischen Standpunkt beschrieben. Die Implementierung ist wichtig um die verschiedenen Vor- und Nachteile dieses Frameworks bewerten zu können, sowie Entscheidungen bei der Umsetzung einer Volltextsuche zu unterstützen. Das Kapitel schließt mit einer Bewertung dieses Frameworks aus softwaretechnischer Sicht.

Die bestehende Architektur (FIA) beschreibt Kapitel 4, wobei der Schwerpunkt auf die für eine Suche relevanten Teile liegt. Die Notwendigkeit einer Volltextsuche wird motiviert und die spezifischen Anforderungen definiert.

Kapitel 5 beschäftigt sich mit der Umsetzung der ermittelten Anforderungen. Zunächst werden verschiedene Alternativen beschrieben und aufgrund intuitiver Vergleichskriterien bewertet. Basierend auf diesem Vergleich wurden zwei Varianten ausgewählt und implementiert. Das Kapitel beschreibt die wichtigsten Elemente der Implementierung, sowie Problemstellungen und deren Lösungen.

Für eine empirische Bewertung der gewählten Varianten und der Tauglichkeit von Lucene werden in Kapitel 6 praktische Tests durchgeführt und deren Ergebnisse beschrieben. Um Messungen auf verschiedenen Netzgrößen vornehmen zu können, wurde ein Modellgenerator implementiert und Modelle mit verschiedenen Eigenschaften erstellt. Auf diesen Modellen wurden die Laufzeiten typischer Anwendungsfälle gemessen. Das Kapitel schließt mit einer Bewertung der ausgewählten Ansätze basierend auf den Ergebnissen.

Kapitel 7 fasst die wichtigsten Ergebnisse dieser Arbeit zusammen und zeigt Ansätze für weitere Forschungen auf.

## 1.2 Verwendete Notation

Für grafische Darstellungen wird soweit möglich die Unified Modelling Language (UML) verwendet. Eine Beschreibung ist in [43] zu finden. Zentrale Bestandteile in den Abbildungen sind zur besseren Übersicht farblich hinterlegt.

Neue Begriffe auf welche später referenziert wird sind ***fett und kursiv*** dargestellt, wobei deren Verwendung nur *kursiv* formatiert ist. Variablen werden ebenfalls *kursiv* dargestellt. Unterstrichene Begriffe werden im Glossar und Abkürzungsverzeichnis (Anhang A ) erklärt. Java-Pakete und Klassen werden durch Arial-Schrift gekennzeichnet. Methoden, Parameter und Quellcode sind in `Courier-Schrift` formatiert.

## Kapitel 2

# Theoretische Grundlagen

Dieses Kapitel beschäftigt sich mit der Problematik der Suche allgemein. Die verschiedenen bekannten Verfahren und Optionen werden zusammengestellt und beschrieben. Aus Platzgründen beschränken sich die Ausführungen auf die für die Arbeit relevanten Bereiche. Eine ausführliche Einführung in die Thematik liefern [7] und [46], welche auch als Basis dieses Kapitels verwendet wurden.

## *2.1 Motivation*

Immer mehr Anwendungen verwalten immer mehr Informationen. Um Informationen gezielt aufzufinden ist eine einheitliche Strukturierung notwendig. Da fast immer verschiedene orthogonale Strukturierungskriterien existieren, ist diese aber oft nicht oder nur mit großem Aufwand möglich. Bei sehr großen Datenmengen hilft auch eine klare Strukturierung nicht für ein schnelles Auffinden der Informationen. Diesem Problem wird mit dem Einsatz von Suchfunktionen begegnet.

## *2.2 Online-Suche vs. Indexgestützte Suche*

Für die Umsetzung einer Suche existieren zwei Grundformen. Die *Online-Suche* arbeitet direkt (online) auf dem Datenbestand während die *indexgestützte Suche* den Datenbestand vor der Suche analysiert und aufbereitet. Diese beiden Varianten werden in diesem Abschnitt vorgestellt, wobei sich die restlichen Ausführungen auf die *indexgestützte Suche* beschränken.

### 2.2.1 Online-Suche

Eine offensichtliche Lösung des Suchproblems ist die sequentielle Analyse des gesamten Datenbestandes. Der Datenbestand wird in keiner Weise für eine Suche vorbereitet und muss online verfügbar sein. Die Online-Suche kann mit dem Durchlesen eines Buches verglichen werden, mit dem Ziel alle Kapitel zu finden, welche einen bestimmten Text enthalten. Dieses Verfahren hat offensichtlich eine Mindest-Aufwandskomplexität von $O(n)$, wobei $n$ die Anzahl der Wörter in den vorhandenen Daten ist. Da String-Vergleiche relativ aufwändig sind, ist dieses Verfahren nur bei sehr kleinen Datenbeständen (z.B. eine einzelne Textdatei) sinnvoll einsetzbar. Zwingend notwendig ist dieses Verfahren, wenn die Daten Änderungen in sehr kurzen Zeitabständen unterworfen sind, oder der Overhead anderer Verfahren nicht akzeptabel ist ([7, p191]).

Für die Online-Suche gibt es eine Reihe bekannter Algorithmen und Verfahren, welche teilweise auch Basis für Indizierungsmechanismen sind. Diese implementieren die exakte Suche, welche wie folgt definiert werden kann:

| | |
|---|---|
| Gegeben: | kurzes Muster $P$ der Länge $m$ |
| | langer Text $T$ der Länge $n$, wobei $n >> m$ |
| Aufgabe: | Finde alle Positionen in $T$ bei denen das Muster $P$ auftritt |

Der Brute-Force Algorithmus ist der einfachste der bekannten Algorithmen. Der Text wird sequentiell gelesen und für jedes Zeichen ein Vergleich der aktuellen Stelle mit *P* durchgeführt. Dies bedeutet, dass für jedes im Text enthaltene Zeichen die jeweils *m* nächsten Zeichen analysiert werden müssen. Das Verfahren kann etwas verbessert werden, indem eine Prüfung abgebrochen wird, sobald ein Zeichen auftritt, welches nicht an der entsprechenden Stelle in *P* enthalten ist.

Der maximale Aufwand setzt sich zusammen aus *O(n)* Textpositionen, welche jeweils mit maximal *O(m)* untersucht werden. Es ergibt sich somit ein WorstCase von *O(mn)*.

Effizientere Algorithmen minimieren den WorstCase auf *O(n)*. Bekannte Vertreter dieser Algorithmen sind Knuth-Morris-Pratt ([38, p. 329ff]) und Boyer-Moore ([38, p. 334ff]). Weitere Algorithmen werden in [7] und [38] beschrieben.

### 2.2.2 Indexgestützte Suche

Für große Datenmengen ist die *Online-Suche* sehr ineffizient, bzw. wie im Fall von Internet-Suchmaschinen praktisch nicht einsetzbar. Die indexgestützte Suche verspricht hier wesentliche Effizienzsteigerungen.

In Fachbüchern ist es üblich, am Ende des Buches einen so genannten Index einzufügen. Dieser bietet eine alphabetisch sortierte Liste aller relevanten im Buch verwendeten Begriffe, zusammen mit Referenzen auf die entsprechenden Seiten. Ein ähnliches Verfahren, auch als Indizierung bezeichnet, kann auch für EDV-gestützte Suchfunktionen verwendet werden. Die Grundidee hier ist, schnelle Suchen durch den Einsatz vorbereitender Massnahmen zu ermöglichen.

Die indexgestützte Suche unterteilt sich in zwei Grund-Schritte:

1. Aus den vorhandenen Daten wird eine für Suchanfragen geeignete Datenstruktur (Index) erstellt. Dieser Schritt wird als "Indizierung" bezeichnet. Dies wird meist vorbereitend ohne konkrete Suchanfrage durchgeführt.

2. Basierend auf einer Suchanfrage werden die Ergebnisse aus einem oder mehreren Indizes ermittelt. Ein Zugriff auf die realen Daten ist nicht notwendig. Dieser Schritt wird als "*indexbasierte Suche*" bezeichnet.

Durch diese Trennung ergibt sich eine Reihe von Vorteilen:

- Unterschiedliche Datenquellen (z.B. Datenbanken, Dateisysteme, gemessene oder berechnete Werte) können über ein gemeinsames Verfahren analysiert werden. Eventuelle Formatkonvertierungen erfolgen nur einmalig bei der Indizierung und nicht bei jeder Suchanfrage.

- Der Datenbestand wird nur einmal komplett analysiert. Suchanfragen können alleine durch Zugriff auf den Index erfolgen. Wird der Index lokal abgelegt, so sind auch bei verteilten Daten keinerlei Netzwerkzugriffe erforderlich. Ebenso können auch Daten in der Suche berücksichtigt werden, welche zum Suchzeitpunkt nicht direkt verfügbar sind (z.B. bei Archivierung in einem Band-Archiv)

- Die Mächtigkeit einer Suche kann auch durch aufwändige Indizierungsmechanismen vergrößert werden, ohne zu einer negativen Beeinträchtigung der Antwortzeit der Suche zu führen. Dies ist darin begründet, dass die Indizierung ohne Benutzerinteraktion erfolgt - meist sogar nur in periodischen Zeitabständen (z.B. einmal täglich). Die hierbei benötigte Zeit kann, ausser bei sehr großen Datenmengen, in der Regel vernachlässigt werden.

- Die Suche über einen Index kann aufgrund der zugeschnittenen Datenstruktur sehr effizient erfolgen. Die Komplexität kann auf die Größenordnung *O(log n)* verringert werden.

Die Indizierung wird gewöhnlich als vorbereitende Maßnahme unabhängig von Suchanfragen durchgeführt. Da die Indizierung nur von den bestehenden Daten abhängt, muss diese auch nur bei Änderungen dieser Daten erfolgen. Meist wird hier ein als "inkrementelle Indizierung" bezeichnetes Verfahren angewandt. Dabei wird zunächst initial auf den bestehenden Daten ein Index aufgebaut. Bei Änderungen der Daten (z.B. neu hinzugekommene, geänderte oder gelöschte Dateien) werden nur diese Änderungen in den Index integriert. Dies erhöht die Performanz beträchtlich, da nicht jedes Mal der gesamte Datenbestand analysiert werden muss.

Unverzichtbar ist die indexgestützte Suche bei sehr großen Datenmengen, wie sie beispielsweise bei Intra- oder Internet-Suchmaschinen vorkommen. Aufgrund des Zeitbedarfs für die Indizierung ist diese Form der Suche allerdings ungeeignet für sich sehr schnell ändernde Daten, wie beispielsweise Börsenkurse.

## 2.3 Begriffsdefinitionen

Es gibt eine Reihe von Suchmethoden, welche teilweise sehr unterschiedliche Einsatzgebiete haben. Diese beschränken sich nicht nur auf Texte, sondern beispielsweise auch auf genetische Datenbanken. Die vorliegende Arbeit beschränkt sich auf die für Texte relevante Methoden. In diesem Abschnitt werden die im Folgenden verwendeten Bezeichnungen definiert.

Ein Objekt des Suchraums wird als ***Dokument*** bezeichnet. Dies umfasst alle Arten von Daten, wie Dokumente im üblichen Sinne (z.B. Artikel einer Zeitschrift), Emails, Log-Dateien oder auch berechnete oder gemessene Daten. Der Inhalt eines Dokumentes kann klassifiziert werden, z.B. nach Autor, Titel und Inhalt.

Die Untereinheit von einem Dokument ist ein ***Wort***. Die Definition schließt nicht nur Wörter im sprachlichen Sinne ein, sondern alle Teilstücke eines Dokuments wie z.B. auch Datumsangaben oder IP-Adressen.

Eine ***Suchanfrage*** besteht aus einer oder mehreren ***Teilanfragen*** und deren Verknüpfungen. Eine Teilanfrage besteht entweder aus weiteren Teilanfragen oder aus einer Termanfrage. Eine ***Termanfrage*** bezeichnet eine Suche nach einem einzelnen Begriff, welcher als ***Term*** bezeichnet wird. Dies kann ein einzelnes Wort, aber auch ein Wort- oder Satzteil sein.

Ein Term ***passt*** auf ein Wort, wenn alle Dokumente welche dieses Wort enthalten durch eine Anfrage nach diesem Term gefunden werden. Das Wort selbst wird als ***passendes Wort*** bezeichnet. Sofern bei mehreren Termen die Verknüpfungen der Suchbedingungen dies nicht verhindern, wird das zugehörige Dokument Bestandteil des ***Suchergebnisses*** sein.

## 2.4 Such-Modelle

Ein Suchmodell definiert die Möglichkeiten der Definition von Suchbedingungen und/oder deren Kombinationsmöglichkeiten. Durch die Wahl des Suchmodells wird somit die Funktionalität einer Suchmaschine zu einem großen Teil festgelegt.

Im Rahmen dieser Arbeit sind die Suchmodelle *Boolean* und *Extended Boolean*, welches auf dem *Vektor-Modell* aufbaut, relevant. Weitere Suchmodelle werden in [7] beschrieben.

### 2.4.1 Boolean-Modell

Das ***Boolean-Modell*** verkettet mehrere Suchanfragen unter Verwendung der booleschen Operatoren AND, OR und NOT. Dies ermöglicht eine zielgerichtete Suche, wenn mehrere Bestandteile oder Eigenschaften der gewünschten Dokumente bekannt sind.

Eine Suchanfrage setzt sich zusammen aus *Termanfragen*, welche Mengen von Dokumenten ergeben und Operatoren, welche diese Mengen kombinieren. Da eine Boolean-Suche wiederum eine Menge von Dokumenten zurückgibt, können diese (zumindest theoretisch) beliebig verschachtelt werden, so dass ein Syntax-Baum aufgebaut wird, dessen Blätter die einzelnen *Termanfragen* sind. Abbildung 2-1 zeigt ein Beispiel für einen solchen Baum.

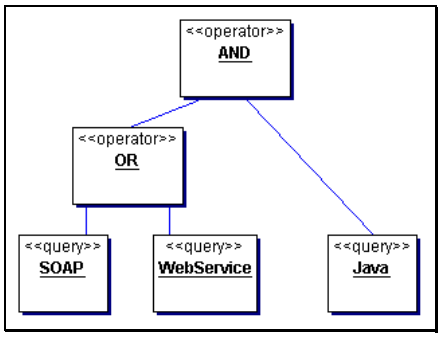


**Abbildung 2-1 Beispiel für einen Syntax-Baum der Boolean-Suche**

Diese Suche findet alle Dokumente welche die Wörter "Java" und "SOAP" oder "Java" und "WebService" enthalten.

Das Boolean-Modell arbeitet auf einer strikten Trennung zwischen relevanten und irrelevanten Dokumenten. Ein Dokument ist also entweder relevant (und damit Teil des *Suchergebnisses*) oder nicht relevant. Der Vorteil liegt in der klaren und einfachen Formalisierung ([7], p. 27).

### 2.4.2 Vektor Modell

Für die Datenmengen, wie sie heute beispielsweise bei Internet-Suchmaschinen auftreten, ist die strenge Suche nach dem *Boolean-Modell*, welche nur ein Dokument entweder findet oder nicht, aber nur bedingt geeignet, da die Ergebnismengen meist zu umfangreich sind. Die Sortierung der Ergebnisse ist dabei zufällig, so dass das gewünschte Dokument vielleicht erst an 1000ster Stelle im Suchergebnis vorkommt.

Um diesem Problem zu begegnen gewichtet das Vektor-Modell die in den Suchanfragen und den Dokumenten vorkommenden Terme (i. d. R. Wörter). Die Gewichtung wird verwendet,

um den Grad der Relevanz der Dokumente zu berechnen. Die Suchergebnisse werden nach dieser Relevanz sortiert, so dass die Dokumente mit der besten Übereinstimmung am Anfang stehen.

Die einzelnen Terme können unterschiedlich gewichtet werden, um deren Wichtigkeit für den Anwender zu reflektieren. Durch den Einsatz der Relevanz können auch Terme angegeben werden, welche nicht zwingend in den Dokumenten vorhanden sein müssen. Dokumente welche diese dennoch enthalten werden höher bewertet. Dieses Modell liefert wesentlich bessere Ergebnisse als das *Boolean-Modell* ([7]).

Die Berechnung der Relevanz wird in Abschnitt 2.5 beschrieben.

### 2.4.3 Extended-Boolean Modell

Ein großer Kritikpunkt am *Boolean-Modell* ist die strikte Anwendung der booleschen Logik, welche, wie zuvor bereits ausgeführt, bei großen Datenmengen zu Problemen führt. Aus diesem Grund verwenden die neueren Suchmaschinen eine Kombination des *Boolean-Modells* mit Elementen aus dem *Vektor-Modell*. Diese Kombination wurde 1983 als ***Extended Boolean Modell*** ([35]) eingeführt.

Die Grundidee ist, alle Dokumente welche mindestens eine der angegebenen Bedingungen erfüllen in das Suchergebnis aufzunehmen, unabhängig von den booleschen Verknüpfungen. Die Verknüpfungen fließen in die Berechnung der Relevanz ein. Tabelle 2-1 stellt dieses Modell dem Boolean-Modell gegenüber.

| | **Terme** | | **Boolean-Modell** | | **Extended Boolean** | |
|---|---|---|---|---|---|---|
| | **A** | **B** | **A or B** | **A and B** | **A or B** | **A and B** |
| **Dokument 1** | 1 | 1 | 1 | 1 | 1 | 1 |
| **Dokument 2** | 1 | 0 | 1 | 0 | $1/\sqrt{2}$ | $1-1/\sqrt{2}$ |
| **Dokument 3** | 0 | 1 | 1 | 0 | $1/\sqrt{2}$ | $1-1/\sqrt{2}$ |
| **Dokument 4** | 0 | 0 | 0 | 0 | 0 | 0 |

**Tabelle 2-1 Boolesche Verknüpfungen beim Boolean- und Extended Boolean Modell ([35])**

Die Tabelle definiert vier Dokumente, mit den Termen *A* und *B* in den vier möglichen Kombinationen (1 steht für enthalten, 0 für nicht enthalten). Die Zellen für die beiden Modelle enthalten die Relevanzwerte für die entsprechenden booleschen Anfragen. Der Unterschied zwischen den beiden Modellen kommt in der Behandlung von Dokumenten, welche nur einen Term enthalten zur Geltung. Die Anfrage "A or B" behandelt im Boolean-Modell die Dokumente 1, 2 und 3 identisch mit einem Relevanzfaktor von 1. Die Anfrage "A and B" behandelt die Dokumente 2, 3 und 4 identisch mit einem Faktor von 0. Das Extended Boolean Modell bewertet dagegen Dokument 1, welches beide Terme enthält, höher als die Dokumente 2 und 3, welche nur einen Term enthalten.

## 2.5 Relevanz

Die Relevanz soll die Korrektheit eines Suchergebnisses in Bezug auf die gewünschten Informationen widerspiegeln. Dies wird erreicht, indem zu jedem Dokument ein Wert zwischen 0 und 1 berechnet wird. Je größer dieser Wert ist, desto größer ist die Übereinstimmung mit den Suchbedingungen. Ob dies auch die Wichtigkeit für den Anwender widerspiegelt hängt natürlich von der Korrektheit der Suchbedingungen ab. In die Berechnung geht eine Vielzahl von Faktoren ein, welche sich von Suchmaschine zu Suchmaschine unterscheiden. Das verwendete Suchmodell kann dabei weitere Faktoren einfügen, wie beispielsweise das zuvor beschriebene *Extended-Boolean-Modell*. Manche Suchmaschinen erlauben dem Anwender zusätzlich die einzelnen Suchbedingungen zu gewichten (auch als *TermWeighting* bezeichnet) und so die Berechnung der Relevanz zu beeinflussen.

Die Faktoren lassen sich in die zwei Gruppen *textabhängige Faktoren* und *dokumentabhängige Faktoren* unterteilen.

Die ***textabhängigen Faktoren*** werden für jeden einzelnen Term der Suchbedingungen berechnet und verwenden als Grundlage ausschließlich den Inhalt des Dokumentes. Ein oft verwendeter Faktor ist beispielsweise die Häufigkeit des Terms in Bezug auf die Gesamtlänge des Textes. Hier steckt die Annahme dahinter, dass viele Vorkommen eine zentrale Rolle dieses Terms im Dokument widerspiegelt. Ein weiterer Faktor kann durch die explizite Auszeichnungen einzelner Wörter im Dokument als Schlüsselwörter definiert werden.

Die ***dokumentabhängigen Faktoren*** spiegeln die Wichtigkeit des gesamten Dokumentes, unabhängig von den einzelnen Termen wider. Es handelt sich dabei um Faktoren, welche an das Dokument angefügt werden und bei Suchen nicht neu berechnet werden müssen. Dieses Verfahren wird auch als *DocumentWeighting* bezeichnet. Im Intranet einer Firma kann dies beispielsweise dafür verwendet werden, um die Wichtigkeit von Dokumenten über firmenweit geltende Richtlinien höher zu bewerten als die von projektspezifischen Richtlinien. Für die automatisierte Berechnung der Wichtigkeit von Dokumenten existieren diverse Verfahren. Ein Beispiel wäre das *Pagerank-Verfahren* ([31]), wie es beispielsweise von Google ([18]) verwendet wird. Dieser verwendet als Kriterium die Anzahl der Links auf ein Dokument und spiegelt dadurch dessen Popularität wider. Dieser Algorithmus ist allerdings umstritten, da hier die Gefahr besteht, dass "auf Dauer die auffälligen und großen Sites überproportional an Aufmerksamkeit gewinnen" ([40]). Dies wird damit begründet, dass sie in den Suchergebnissen weit oben erscheinen und damit größere Chancen erhalten von Autoren verlinkt zu werden, wodurch deren PageRank weiter steigt.

## 2.6 Optionen bei der Indizierung

Die Mächtigkeit einer Suchmaschine kann durch eine Reihe optionaler Maßnahmen beeinflusst werden. Diese Maßnahmen setzen bei der Indizierung durch eine Aufbereitung des Textes an. Die Suchbedingungen müssen meist auf dieselbe Weise aufbereitet werden um die Kompatibilität zu gewährleisten. Im Folgenden werden einige gebräuchliche Optionen beschrieben.

### 2.6.1 CaseFolding

Meistens ist für eine Suche die Groß-Kleinschreibung nicht relevant. So sollen Suchen nach dem Wort "testen" im Normalfall alle Dokumente zurückgeben, welche den Satz "Die Skripte testen die Erreichbarkeit" oder den Satz "Testen ist eine für Informatiker unbeliebte Disziplin"

enthalten (***case-insensitive Suche***). Es gibt aber durchaus auch Fälle, in welchen die Groß-Kleinschreibung beachtet werden soll. (***case-sensitive Suche***) Dies ist insbesondere bei Eigennamen und Abkürzungen der Fall. Eine case-insensitive Suche nach der Abkürzung "SOAP" (**S**imple **O**bject **A**ccess **P**rotocol) würde nicht nur die gewünschten Dokumente über WebServices sondern auch Dokumente zur Geschichte der Seife (englisch: "soap") zurückgeben.

Werden keine case-sensitiven Suchen benötigt, so ist der einfachste Weg, alle Texte vor der Indizierung entsprechend aufzubereiten. Dies hat den Vorteil einer Indexverkleinerung, da "testen" und "Testen" nicht als zwei Wörter abgelegt werden. Des Weiteren muss der Index keine speziellen Maßnahmen für case-insensitive Suchen unterstützen. Bei der Definition von Suchanfragen müssen diese ebenfalls aufbereitet werden.

Müssen sowohl case-sensitive als auch case-insensitive Suchen unterstützt werden, so gibt es prinzipiell zwei Varianten. Entweder wird der Text sowohl in seiner Ursprungsform als auch entsprechend aufbereitet indiziert (wodurch allerdings die Indexgröße praktisch verdoppelt wird), oder der Index unterstützt derartige Mischanfragen direkt. Ein Beispiel für solch einen Index ist laut ([46] p. 146) die *New Zealand Digital Library* (NZDL, [44]).

### 2.6.2 Stemming (Wortstammsuche)

Die Wortstammsuche hat zum Ziel die verschiedenen Ausprägungen eines Wortes einheitlich zu behandeln. So sollen beispielsweise Suchen nach "Bedingung" auch Dokumente mit dem Wort "bedingt" oder "Vorbedingung" finden.

Erreicht wird dieses Ziel, indem alle Wörter eines Textes im Index durch ihren Wortstamm (im Beispiel "beding") ersetzt werden. Im Falle einer korrekten Ersetzung werden sowohl Vor- als auch Nachsilben entfernt. Im Kontext der Indizierung ist es nicht notwendig, dass die resultierenden Wörter sinnbringende Wörter sind, da der aufbereitete Text nicht direkt gelesen wird. Allerdings muss die Abbildung injektiv sein, so dass keine Wörter mit unterschiedlichen Stämmen auf denselben Text abgebildet werden.

Eine korrekte automatische Ermittlung der Wortstämme ist aufgrund der in fast jeder Sprache existierenden Spezialfälle nur unter Verwendung eines Wörterbuches möglich. Da dies jedoch einen enormen Aufwand bedeutet, werden bei der Indizierung meist heuristische Regeln verwendet und kleinere Fehler in Kauf genommen.

Beispiele für solche Algorithmen sind der Algorithmus von Martin Porter ([33], englische Sprache) und der Algorithmus von Jörg Caumanns ([8], deutsche Sprache), welche allerdings beide keine Vorsilben entfernen.

Die angesprochenen Fehler sind beispielsweise beim Algorithmus von Caumanns bei der Verarbeitung von Fremdwörtern zu sehen. So werden die Wörter "Häuser" und "Haus" beide korrekt zu "hau" reduziert, die Wörter "Kompressor" und "Kompression" dagegen nicht zu "kompress", sondern zu "kompressor" und "kompressio".

Die Wortstammersetzung ist jedoch nicht für alle Texte bzw. Textbestandteile sinnvoll. So sollte z.B. die Autorangabe von Dokumenten nicht ersetzt werden ([46, p.146f]).

Verschiedene Alternativen für den Aufbau dieser Algorithmen sind in [7, p. 168f] zu finden.

### 2.6.3 Phonetische Suche

Die phonetische Suche hat zum Ziel, ähnlich klingende Wörter gleich zu behandeln. So sollen Suchen nach "Maier" auch "Meier" oder "Mayer" finden. Diese Funktion wird beispielsweise bei den elektronischen Telefonbüchern der Auskunft verwendet, bei denen der Anwender ein gesprochenes Wort als Suchbegriff eingibt. Die Internet-Variante auf [9] bietet in der Detail-Suche ebenfalls diese Möglichkeit.

Eine verbreitete Variante, auch *Soundex* genannt, transformiert buchstabenbasiert jedes Wort in eine kanonische Form, wobei der erste Buchstabe beibehalten wird. Der Algorithmus ist im Folgenden dargestellt:

1. Ersetze jeden Buchstaben, mit Ausnahme des Ersten, durch den zugehörigen phonetischen Code (siehe Tabelle 2-2)

2. Eliminiere alle adjazenten Duplikate

3. Eliminiere alle Stellen mit dem Code *0*

4. Gebe die ersten vier Zeichen des Strings zurück

Tabelle 2-2 zeigt die phonetischen Codes und die zugehörigen Buchstaben.

| phonetischer Code | Buchstaben |
|---|---|
| 0 | aeiouyhw |
| 1 | bpfv |
| 2 | cgjkqsxz |
| 3 | dt |
| 4 | l |
| 5 | mn |
| 6 | r |

**Tabelle 2-2 Phonetische Codes von Soundex**

In den Index werden anstelle der Wörter die phonetischen Code-Darstellungen eingefügt.

Der Algorithmus transformiert allerdings durchaus auch nicht gleich klingende Wörter in den gleichen Code. So werden die Wörter "catherine" und "cotroneo" beide in den Code "c365" transformiert. Ausserdem werden auch gleich klingende Wörter in unterschiedliche Codes umgewandelt, wenn der Unterschied im ersten Buchstaben liegt. So werden die identisch klingenden Worte "kodieren" und "codieren" unterschiedlich in "k365" und "c365" transformiert.

Genauere Methoden, wie beispielsweise phonometrische Verfahren werden in [47] vorgestellt.

### 2.6.4 StopWords

Eine weiteres Verfahren, welches hauptsächlich zur Reduktion der Indexgröße dient, sind ***StopWords***. Die Idee hierbei ist es, Wörter welche sehr häufig vorkommen und für Suchen meist irrelevant sind, gar nicht erst im Index abzulegen. Diese Wörter umfassen meist alle Arten von Füllwörtern wie Artikel (der, eine), Adverben (hier, da), Präpositionen (in, auf), Pronomen (ich, du) und Konjunktionen (und, oder). Bei diesem Verfahren wird unterstellt,

dass niemand nach diesen Wörtern suchen möchte, da sie mit hoher Wahrscheinlichkeit in einer Vielzahl von Dokumenten vorkommen. Problematisch sind hierbei allerdings Wörter, welche nicht nur als Füllwörter sondern auch als Eigennamen verwendet werden. Ein Beispiel wäre das Wort *"heute"*, welches neben der Bedeutung *"dieser Tag"* auch den Namen einer Nachrichtensendung kennzeichnet. Wird dieses Wort als StopWord eingesetzt, so sind Suchen nach Dokumenten über die Nachrichtensendung nicht möglich

Über 38% der TREC werden nach [46] durch nur 33 Wörter gebildet. Diese Wörter nicht zu indizieren hat dementsprechend einen großen Einfluss auf die Indexgröße.

Die zu ignorierenden Wörter werden üblicherweise in einer so genannten ***StopList*** verwaltet. Jedes Wort im Text und auch die Suchbedingungen müssen gegen diese Liste verglichen werden und Übereinstimmungen werden entsprechend eliminiert. Diese Liste kann entweder statisch aufgebaut oder durch einen Algorithmus dynamisch aus dem Text erzeugt werden.

Bei dynamisch erzeugten *StopLists* ist die Vorhersagbarkeit ein großes Problem. So ist die Menge der gültigen Suchbegriffe abhängig vom jeweiligen einzelnen Dokument. Ferner können - je nach Schreibstil - auch die Kernbegriffe verloren gehen. Enthält ein Artikel über eine Firma beispielsweise sehr oft den Firmennamen, so kann passieren, dass genau dieser Name auf die *StopList* gesetzt wird. Als Resultat lassen Suchen nach dem Namen dieser Firma gerade das Dokument aus, welches die Geschichte der Firma beschreibt.

Aber auch statische StopLists können erfolgreiche Suchen verhindern. Ein Beispiel wäre hier der bekannte Ausspruch aus Shakespeares Hamlet "Sein oder nicht sein? ", welcher nur aus potentiellen Kandidaten für eine StopList besteht.

## 2.7 Varianten der Suche

Auch die Suche bietet verschiedene Varianten, welche i. d. R. nicht bei der Indizierung besonders berücksichtigt werden müssen. Eine Kompatibilität der Indexstruktur ist natürlich eine Voraussetzung. Dieser Abschnitt beschreibt einige übliche Varianten.

Die ***Phrasen-Suche*** dient dem Auffinden von Dokumenten mit bestimmten Satzteilen. Oft wird diese Suche eingesetzt, wenn der Titel eines Dokumentes bekannt ist. Phrasen-Suchen können aber auch auf dem Inhalt des Dokumentes sinnvoll sein, beispielsweise wenn der Anwender eine bestimmte Formulierung im gesuchten Dokument kennt.

Die ***Abstands-Suche*** ist eine Erweiterung der Phrasen-Suche und definiert eine maximale Anzahl an Wörtern zwischen den angegebenen Termen. Sie eignet sich beispielsweise um näherungsweise verschiedene Wörter zu finden, welche im selben Satz oder Absatz vorkommen. Die Abstands-Suche kann bei der Verwendung von Eliminierungs-Techniken bei der Indizierung (z.B. StopWords) auch zur Realisierung einer Phrasen-Suche verwendet werden. Die wissenschaftliche Suchmaschine CiteSeer ([30]) verwendet beispielsweise diesen Ansatz wenn in einer Phrasen-Suche StopWords enthalten sind. So wird bei der Phrasensuche "System of Two Spinning Disks" beispielsweise das StopWord "of" eliminiert und die Suche erfolgt nach dem Term "System" und der Phrase "Two Spinning Disks" wobei ein anderes Wort dazwischen vorkommen darf (Abstand 1). Dieses Verfahren bildet allerdings die Phrasensuche nicht vollständig ab, da die Suchanfrage "System for Two Spinning Disks" dieselben Ergebnisse liefert. In der Realität dürften dadurch allerdings nur in Einzelfällen falsche Ergebnisse produziert werden.

Die ***Intervall-Suche*** verwendet zwei Terme, welche die Grenzen eines Intervalls definieren. Alle Wörter in diesem Intervall werden als *passende Wörter* betrachtet. Diese Form der Suche wird meist für temporale oder monetäre Werte verwendet und erlaubt Anfragen wie "Alle Veröffentlichungen im letzten Jahr". Durch die Belegung der Grenzen durch Minimal- bzw. Maximal-Werte sind auch Suchen wie "Alle Veröffentlichungen vor dem 1.1.2003" oder "Alle Veröffentlichungen der letzten 7 Tage" möglich.

Die ***Fehlertolerante Suche***, auch als ***Fuzzy-Suche*** bezeichnet, erlaubt einzelne Abweichungen zwischen den Termen und den Wörtern. Diese Form der Suche eignet sich insbesondere für Suchen auf fehlerbehafteten Dokumenten, wie sie beispielsweise durch Schreibfehler oder optische Zeichenerkennung (OCR) entstehen. Oft wird hier als Basis die ***Levenshtein-Distance*** (auch ***Edit-Distance*** genannt) verwendet. Diese Metrik gibt die minimal nötige Anzahl von Einfügungen, Löschungen und Änderungen an um ein Wort in ein anderes Wort zu überführen. Bei einer solchen Suche wird für jeden Term ein Grenzwert für die *Edit-Distance* angegeben. Alle Wörter mit einer geringeren *Edit-Distance* werden als *passende Wörter* betrachtet. Verschiedene Algorithmen zur Berechnung dieser Metrik sind in [24], Erweiterungen in [36] und Indizierungsvarianten in [29] zu finden.

Die ***Wildcard-Suche*** erlaubt die Angabe von Platzhaltern in Termen. Der Platzhalter "?" steht dabei für ein einzelnes beliebiges Zeichen und der Platzhalter "*" für eine beliebige Anzahl beliebiger Zeichen. Der Term "te?t" passt beispielsweise sowohl auf das Wort "test" als auch auf die Wörter "text" und "tent" (engl.: Zelt), wobei der Term "te*t" auf alle Wörter passt, welche mit "te" anfangen und mit "t" enden, wie das Wort "teeblatt". Zu Beachten ist, dass die Wildcards auch Leerzeichen akzeptieren, so dass der Term "te*t" auch auf "teures tablett" passt.

Die ***Präfix-Suche*** ist ein Spezialfall der *Wildcard-Suche* und ermöglicht die Suche nach Wortanfängen (Präfixe) unter Vernachlässigung der Endungen (Suffixe). Um eine Präfix-Suche auf eine Wildcard-Suche abzubilden wird der Platzhalter "*" als letztes Zeichen des Terms angegeben. Der Term "Baum*" passt sowohl auf die Wörter "Baum" als auch "Baumarkt" oder "Baumaschine". Die in Abschnitt 2.6.2 beschrieben Wortstammsuche unter Verwendung der Algorithmen von Porter und Caumanns können, da keine Vorsilben entfernt werden, durch eine Präfix-Suche realisiert werden. Die Präfix-Suche kann auf eine Intervall-Suche abgebildet werden, indem im Index im ersten Schritt das erste und letzte Vorkommen eines Wortes mit dem angegebenen Anfang gesucht wird und anschließend eine Intervallsuche mit den beiden Wörtern als Termen erfolgt.

Die ***Regular-Expressions-Suche*** bietet durch die Möglichkeit beliebige Muster (reguläre Ausdrücke) zu definieren eine wesentlich mächtigere Definition von Termen als die *Wildcard-Suche*. So kann beispielsweise anstatt des Platzhalters "?", welcher in der *Wildcard-Suche* beliebige Zeichen akzeptiert, eine Einschränkung auf bestimmte Buchstaben erfolgen (z.B. durch [aeiou] auf kleingeschriebene Vokale). Des Weiteren sind aber auch komplexere Muster möglich, wie "-?[0-9]+(,[0-9]*)?", welches negative und positive Zahlen mit und ohne Fließkommateil findet. Zu Beachten ist, dass die Zeichen "?" und "*" in regulären Ausdrücken eine andere Bedeutung haben als in der Wildcard-Suche, so dass bei einer Verwendung beider Such-Möglichkeiten eine explizite Deklaration erfolgen muss, welche Form der Suche anzuwenden ist. Die Mächtigkeit regulärer Ausdrücke ist sehr hoch weswegen eine vollständige Beschreibung den Rahmen dieser Arbeit sprengen würde. Eine Einführung in die Möglichkeiten und die Verwendung regulärer Ausdrücke bietet z.B. [14].

Die ***Synonym-Suche*** findet auch Wörter mit identischer Bedeutung (Synonyme). So könnte eine Suche nach dem Wort "Handy" auch Texte welche das Wort "Mobiltelefon" enthalten zurückgeben.

## 2.8 Index-Strukturen

Dieser Abschnitt beschreibt zwei Möglichkeiten für die Struktur eines Index. Weitere Möglichkeiten, wie Suffix-Bäume und Signatur-Dateien, werden in [7] beschrieben.

### 2.8.1 Invertierter Index

Ein Invertierter Index ist eine Datenstruktur, welche für sehr performante Suchen nach einzelnen Wörtern ausgelegt ist. Im Gegensatz zum Originaltext, welcher nach Wort-Positionen sortiert ist, wird der Index nach den Wörtern sortiert und verweist auf die Positionen.

Abbildung 2-2 zeigt einen Beispieltext und den zugehörigen Index. Der Text wurde zuvor in Kleinbuchstaben umgewandelt. Der Index besteht aus einer Menge von Wörtern, wobei jedes Wort mit einer Liste der Vorkommen verknüpft ist.

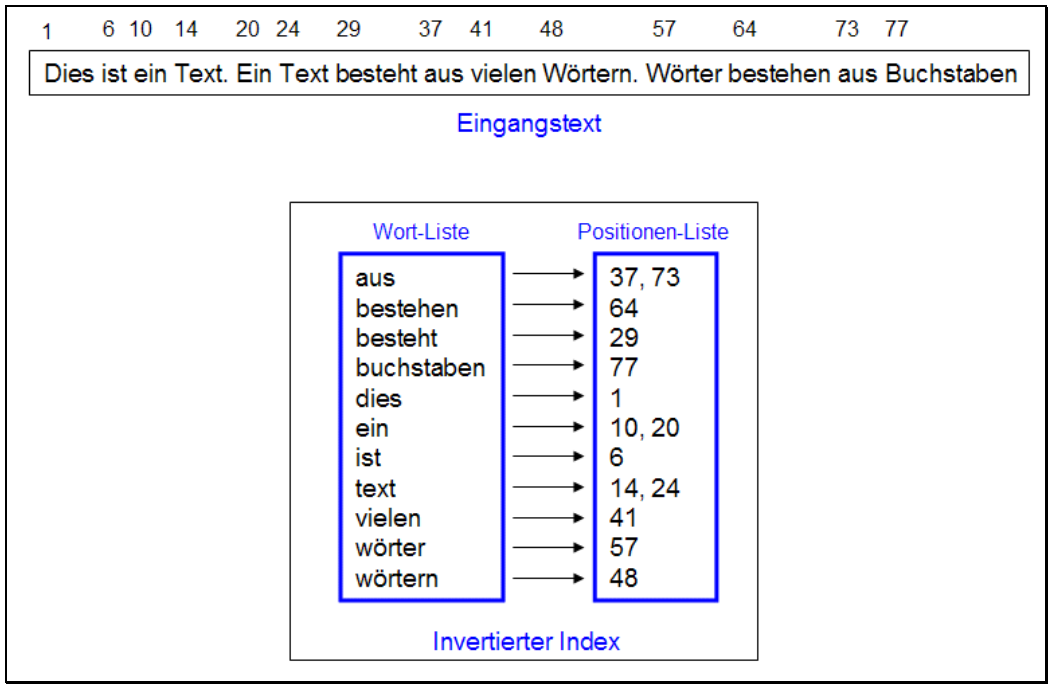


**Abbildung 2-2 Beispieltext mit zugehörigem Invertiertem Index**

Für die Ablage der Textpositionen existieren mehrere Varianten. Im Beispiel wurden die Byte-Positionen der Anfangsbuchstaben verwendet. Dies ermöglicht den direkten Zugriff auf die Wörter im Text. Eine alternative Verwendung der Wort-Position (Wort 1, Wort 2, ...) ermöglicht dagegen eine einfache Implementierung von Phrasen- und Abstandssuchen.

Der Platzbedarf für das Vokabular ist aufgrund der hohen Wahrscheinlichkeit identischer Terme im Vergleich zum Originaltext bei großen Datenmengen relativ gering. Gemäß Heaps Law ([23, p 206ff]) wächst der Platzbedarf für den Index bei englischen Texten nur mit $O(n^{\beta})$, wobei $0 < \beta < 1$. $\beta$ ist abhängig vom zugrunde liegenden Text und liegt nach [7] in der Praxis

zwischen 0,4 und 0,6. Dies kann durch Maßnahmen, wie die Anwendung von *StopWords* (Abschnitt 2.6.4) oder *Stemming* (Abschnitt 2.6.2) weiter reduziert werden.

Die Positionen-Liste hat eine Platzkomplexität von *O(n)*, da für jedes Vorkommen ein Eintrag vorhanden ist. Bei Anwendung von *StopWords* wird dies in der Praxis auf 30% - 40% der Textgröße reduziert ([7]). Eine weitere Reduktion lässt sich durch ein Block-Adressierungsverfahren erreichen. Dabei wird der Ursprungstext in Blöcke eingeteilt und anstatt der Wort-Positionen nur die Blocknummer gespeichert. Ist ein Wort mehrfach in einem Block vorhanden, so wird nur eine einzelne Referenz abgelegt. Die Reduktion erfolgt also durch Verminderung der Anzahl der Referenzen und die Verkleinerung der dafür benötigten Zahlen. Dieses Verfahren wird beispielsweise bei der Suchmaschine Glimpse ([28]) eingesetzt. Der Nachteil dieses Verfahrens ist, dass eine Abstandssuche nicht direkt über den Index erfolgen kann, da die Reihenfolge der Wörter in einem Block nicht im Index abgelegt ist. Wird die exakte Position der Fundstelle benötigt (z.B. zur Hervorhebung bei der Ausgabe), so muss der entsprechende Block nach der Suche nochmals online analysiert werden.

Die Performanz und Möglichkeiten der Suche werden durch die Methode der physischen Ablage beeinflusst. Für reine Einzelwort-Anfragen können alle passenden Strukturen verwendet werden, wie beispielsweise Hashing, Tries oder B-Bäume. Präfix- und Intervall-Anfragen sind dagegen mit Hashing nicht möglich ([7, p. 195]). Eine Wildcard-Suche mit dem Operator "*" wie "*A*B*" ist nur möglich wenn *B* ein Präfix bezeichnet. Legt *B* dagegen kein Präfix fest, so müssen zunächst alle Wörter im Index mit *B* als Suffix gesucht werden, da der Index nach Präfixen sortiert ist.

### 2.8.2 Q-Gram Index

Der ***Q-Gram Index*** ist ähnlich aufgebaut wie der Invertierte Index, jedoch werden die Wörter nicht in ihrer Ursprungsform abgelegt, sondern jeweils alle Wortteile der (festen) Länge *q* (***Q-Gramme***). Zu jedem Q-Gramm sind die Positionen im Text in aufsteigender Sortierung abgelegt. Terme der Länge $l < q$ können direkt aufgefunden werden. Längere Terme werden ebenfalls in Q-Gramme aufgespalten und einzeln im Index gesucht. Anschließend werden die gefunden Textpositionen miteinander verglichen und daraus das Ergebnis berechnet. Abbildung 2-3 illustriert dies an einem Beispiel.

Aus dem Text "Ein Beispiel für einen Text" werden alle möglichen Q-Gramme der Länge 3 extrahiert und in den Index zusammen mit der Position im Text eingefügt. Ein Leerzeichen wird in der Abbildung durch einen Strich dargestellt.

Die Suchanfrage "beispiel" wird in die drei Q-Gramme "bei", "spi" und "iel" zerlegt und deren Abstände ermittelt (+3, +2). Da eine Suche nach einem ganzen Wort erfolgt, werden hier keine Leerzeichen berücksichtigt. Die drei Q-Gramme werden dann einzeln im Index gesucht und die Positionen ermittelt (5, 8 und 10). Die Ergebnisse werden nun mit den Abständen verglichen. Da 5+3=8 und 8+2=10 beginnt das Wort an der Position 5 ein *passendes Wort*.

Die Suche nach einem einzelnen Wort ist offensichtlich aufwändiger als bei dem in Abschnitt 2.8.1 beschriebenen Inverted-File-Index. Die Wildcard-Suche "bei*ext" ist dagegen weitaus effizienter, da hier das Suffix direkt gesucht werden kann und nur ein Größenvergleich erfolgen muss.

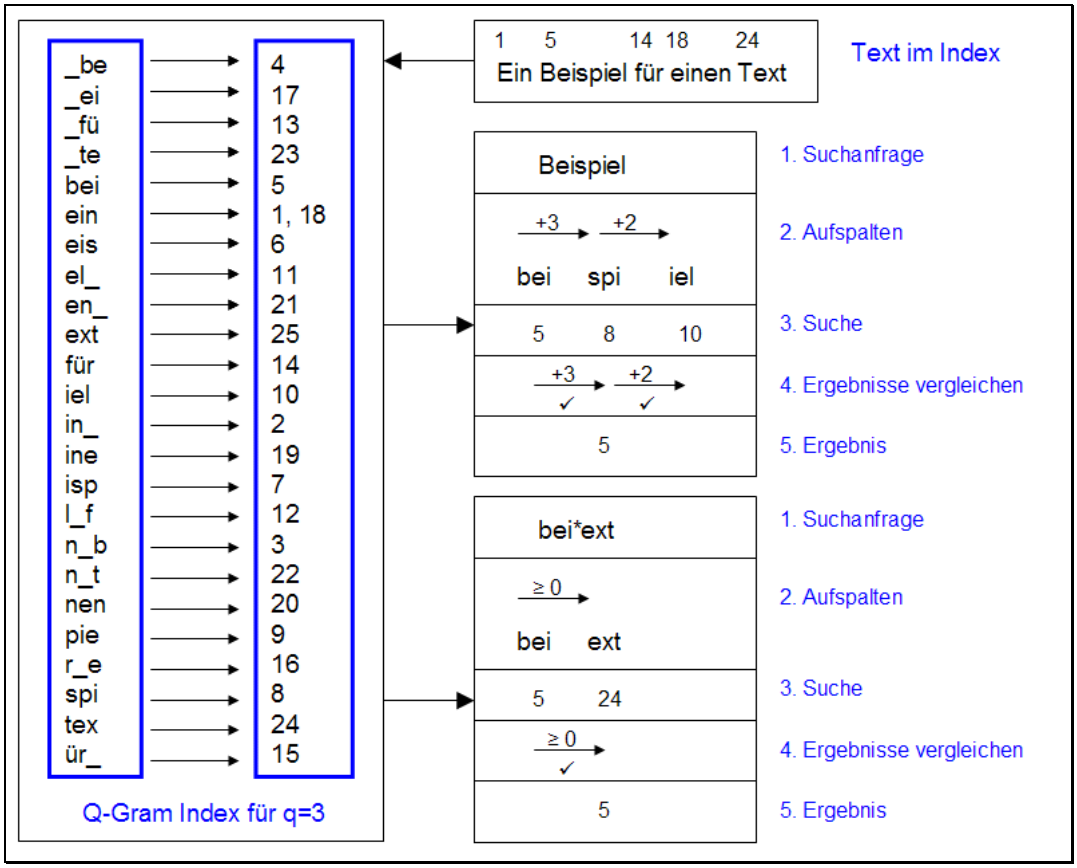


**Abbildung 2-3 Aufbau und Suche im Q-Gram Index**

## Kapitel 3

# Lucene - ein Framework zur Volltextsuche

Dieses Kapitel stellt das Framework "Lucene" ([1]) der Apache Software Foundation (ASF) vor und beschreibt dessen Möglichkeiten. Basis dieses Kapitels ist die Version 1.2. Ziel ist es, dem Leser einen Überblick über die Arbeitsweise, sowie eine Grundlage für die Anwendung in eigenen Projekten, zu geben.

Ein Tutorial für eine einfache Anwendung ist in [22] zu finden, wobei eine Konzentration auf die typische Anwendung erfolgt. Hintergrundinformationen über die genaue Arbeitsweise werden kaum gegeben. Ein weiteres, aber noch oberflächlicheres Tutorial bietet [17], nennt aber auch Auswahlkriterien für Suchmaschinen. Die Indizierung wird in [20] eingeführt, [19] untersucht verschiedene Einfluss-Faktoren auf die Performanz der Indizierung. Weitere Informationen sind auf der Lucene-Webseite ([1]) zu finden.

Dieses Kapitel fasst die oben genannten Quellen zusammen und erweitert gegebenenfalls diese durch weitere Quellen und eigene Versuche. Zunächst wird ein Überblick über die wichtigsten Eigenschaften von Lucene gegeben. Es folgt eine grobe Beschreibung des Ablaufs von den Eingangsdaten bis zur Ausgabe der Suchergebnisse. Anschließend wird jeder Schritt detailliert beschrieben, wobei der Schwerpunkt jeweils zunächst beim Konzept und anschließend bei dessen Implementierung liegt. Zum Schluss wird das Framework aus Sicht des Autors dieses Dokuments bewertet.

## *3.1 Überblick*

Lucene ist ein plattformunabhängiges Framework für den Aufbau von Suchmaschinen. Die Framework-Eigenschaft grenzt es von den meisten kommerziellen Suchmaschinen (wie z.B. Verity Information Server [45]) ab. Die kommerziellen Produkte sind meist vorkonfiguriert und in ihrer Erweiterbarkeit stark eingeschränkt, dafür allerdings ohne weitere Implementierung direkt einsetzbar (sog. "out-of-the-box-Ansatz").

Eine kurze Aufstellung über die Eigenschaften von Lucene ist in [37] zu finden. An dieser Stelle werden nur die für diese Arbeit relevanten Punkte genannt.

Das Framework ist durch die Implementierung in Java plattformunabhängig. Die meisten Komponenten können ausgetauscht werden, wodurch das Framework universell einsetzbar ist. Die Daten werden bei der Indizierung in frei konfigurierbare Felder (z.B. Titel, Autor, Erstellungsdatum etc.) klassifiziert, welche später dann gezielt durchsucht werden können. Durch entsprechende Erweiterungen können beliebige elektronische Daten indiziert werden, Parser für PlainText und HTML sind bereits mitgeliefert. Der Index basiert auf der Klasse der *Invertierten Indizes* (Abschnitt 2.8.1). Die Suche basiert auf dem *Extended Boolean Modell* (Abschnitt 2.4.3) und sortiert demnach die Ergebnisse nach Relevanz (Abschnitt 2.5). Die Suchmöglichkeiten umfassen *Wörter*, *Phrasen*, *Abstandssuche*, *Intervalle*, *Wildcards* und *fehlertolerante Suchen* (siehe Abschnitt 2.7).

Im Folgenden wird nun die Arbeitsweise von Lucene beschrieben und die einzelnen Bestandteile vorgestellt.

## 3.2 Ablauf

Die meisten Suchmaschinen unterteilen die Problematik meist nur in die Schritte "Indizierung" und "Suche" ein ([22]). Lucene nimmt hier eine weitere Detaillierung gemäß den zu lösenden Einzelaufgaben vor:

Indizierung:

1: Aufbereitung des Suchraums

2: Indizierung des Suchraums

Suche:

3: Anfragebearbeitung

4: Suche

5: Ergebnispräsentation

Das Framework wird anhand dieser Schritte beschrieben.

Abbildung 3-1 zeigt eine Grob-Übersicht über den Ablauf und definiert wichtige Begriffe für das Konzept.

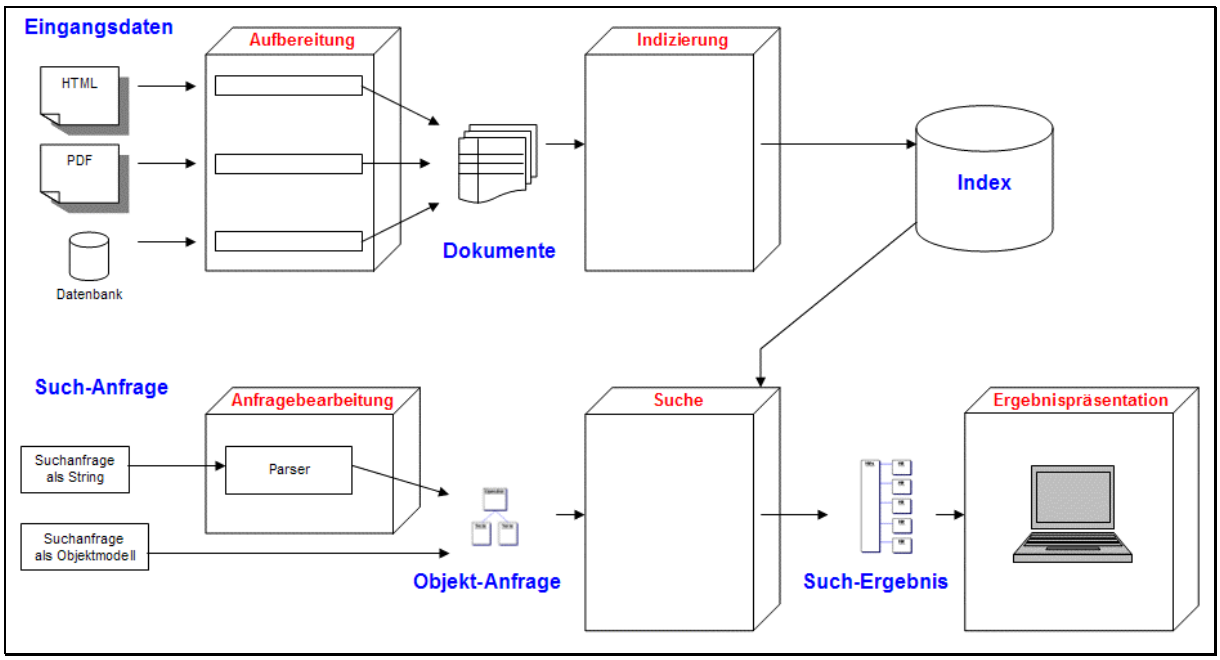


**Abbildung 3-1 Übersicht über die Arbeitsweise von Lucene**

Die *Eingangsdaten* können in beliebigen elektronischen Formaten vorliegen.

Der Schritt ***Aufbereitung*** überführt die *Eingangsdaten* in *Dokumente*. Dabei werden die Daten in Felder (z.B. Autor, Titel, Inhalt, Erstellungsdatum) unterteilt. Die einzelnen Felder werden durch *Dokument-Objekte* aggregiert.

Der Schritt ***Indizierung*** erstellt aus *Dokumenten* einen *Index*. Dabei werden die *Dokumente* aufbereitet um verschiedene Optionen für die Suche zu ermöglichen (z.B. Entfernung von *StopWords* (Abschnitt 2.6.4), *Stemming* (Abschnitt 2.6.2), *CaseFolding* (Abschnitt 2.6.1) etc.).

Der Schritt ***Anfragebearbeitung*** erzeugt aus einer *Suchanfrage* eine ***Objekt-Anfrage***. Eine ***Suchanfrage*** kann entweder als String vorliegen (z.B. durch Eingabe in einer Oberfläche) oder bereits als *Objekt-Anfrage* (bei Komposition durch die Anwendung). In diesem Fall ist der Schritt *Anfragebearbeitung* nicht notwendig.

Der Schritt ***Suche*** ist der zentrale Schritt. Hier wird basierend auf einer *Objekt-Anfrage* ein *Such-Ergebnis* erzeugt. Hierzu werden die Daten im *Index* verwendet.

Der Schritt ***Ergebnispräsentation*** dient der Darstellung des *Such-Ergebnisses* in einem für den Anwender oder die Anwendung geeigneten Format. Dies kann z.B. durch die Ausgabe einer HTML-Seite erfolgen.

Die einzelnen Schritte werden in den Abschnitten 3.3 bis 3.12 ausgeführt.

## 3.3 Aufbereitung des Suchraums - Konzept

Im ersten Schritt werden die *Eingangsdaten* in ein für die Indizierung geeignetes Format (***Dokumente***) überführt. Viele Suchmaschinen indizieren nur Dateien. Lucene bietet dagegen eine offene API welche durch Erweiterung die Indizierung beliebiger Daten erlaubt. So können - je nach Implementierung - beispielsweise auch Datenbank-Tabellen, Inhalte von Zip-Archiven oder auch berechnete Daten indiziert werden.

Die Eingangsdaten bestehen üblicherweise aus einem Text (z.B. Dateiinhalt) und einer Reihe von Meta-Informationen (z.B. Autor, Dateiname, Objekt-Referenz, ...). Prinzipiell kann jede dieser Informationen zur Suche herangezogen werden. Meist ist für die Suche aber nur eine bestimmte Untermenge relevant, und eine andere Untermenge für die Präsentation der Ergebnisse. Lucene begegnet dieser Trennung durch eine Klassifizierung der Meta-Informationen in beliebig definierbare Felder. Die Klassifizierung beinhaltet ob ein Feld durchsuchbar (***index***) ist und ob dessen Inhalt im Index unverändert gespeichert (***store***) werden soll. Bei durchsuchbaren Feldern kann weiterhin festgelegt werden, ob diese aufbereitet werden dürfen (***token***). Die Aufbereitung betrifft die Anwendung von Optionen (Abschnitt 2.6), wie z.B. *StopWords* oder *CaseFolding*. Nicht alle Klassifizierungen sind sinnvoll und zulässig. Tabelle 3-1 stellt die typischen Kombinationen dieser Parameter zusammen.

| store | index | token | Bemerkung / typische Verwendung |
|---|---|---|---|
| nein | ja | ja | Daten welche nur durchsucht werden sollen,<br><br>z.B. Dateiinhalt |
| ja | nein | nein | Daten welche nur zur Ausgabe benötigt werden,<br><br>z.B. Pfad zu einer Datei |
| ja | ja | nein | Daten welche unverändert durchsucht und ausgegeben werden können,<br><br>z.B. Schlüsselwörter, durchsuchbare Referenzen, Datumsangaben |
| ja | ja | ja | Daten welche durchsucht und ausgegeben werden sollen,<br><br>z.B. Abstract eines Artikels |

**Tabelle 3-1 Typische Kombinationen von Parametern der Felder**

Durch die Aufteilung der Daten in verschiedene Felder werden wesentlich komplexere Abfragen wie z.B. "Autor heißt Müller und Text enthält Datenbank" ermöglicht. Gleichzeitig wird der Index von irrelevanten Daten befreit und Optionen bei der Suche können auf einzelne Felder ausgerichtet werden.

Sollen *Dokumente* aktualisiert werden können, so sollte ein eindeutiger Identifier als Schlüsselwort-Feld angefügt werden, da Löschungen im Index durch Angabe einer Suchanfrage erfolgen. Dieser Identifier kann durchaus auch in mehreren Dokumenten verwendet werden, wenn diese eine logische Einheit bilden und nur zusammen aktualisiert oder gelöscht werden.

Die Aufbereitung dient insbesondere der Konvertierung verschiedener Datei- und Datenformate in ein einheitliches Modell, sowie der Definition der durchsuchbaren Daten. Lucene liefert Implementierungen zur Auswertung von Text- und HTML-Dateien mit. Auf der Webseite ([1]) sind Links zu Implementierungen für XML, PDF und RTF vorhanden.

Die Auswahl der zu indizierenden Daten muss durch die Anwendung erfolgen. Im Gegensatz zu anderen Suchmaschinen (z.B. Verity Information Server [45]) liefert Lucene keinen WebCrawler oder ähnliches mit.

Für den Nutzen einer Suchmaschine ist die Auswahl der Daten der wichtigste Schritt, da er die Funktionalität und Mächtigkeit der Suche zu einem großen Teil definiert.

## 3.4 Aufbereitung des Suchraums - Implementierung

Ein Dokument wird durch Instanzen der Klasse org.apache.lucene.document.Document repräsentiert. Im Wesentlichen handelt es sich dabei um eine dynamische Liste von Name/Wert-Feldern, welche durch Instanzen von org.apache.lucene.document.Field dargestellt werden. Abbildung 3-2 zeigt diese Klassen in UML-Darstellung.

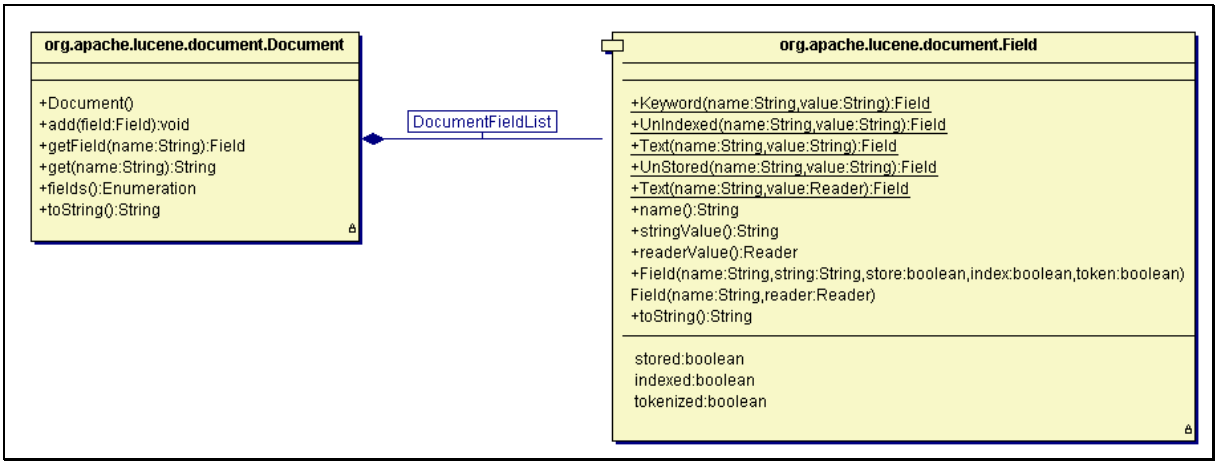


**Abbildung 3-2 Die Klassen Document und Field**

Die Field-Instanzen klassifizieren die Informationen und legen damit die weitere Verarbeitung fest. Der öffentliche Konstruktor der Field-Klasse erwartet hierzu die drei bereits genannten booleschen Parameter `store`, `index` und `token`. Die Übergabe der Feld-Inhalte kann entweder als String oder als ein java.io.Reader-Objekt erfolgen. Ein Reader-Objekt wird insbesondere zur Übergabe von Dateiinhalten verwendet, um die sonst nötige teure Zwischenspeicherung der gesamten Datei als String zu vermeiden. Nicht alle denkbaren Kombinationen der Klassifizierungsparameter sind zulässig und sinnvoll. Falsche Aufrufe werden über den Java-Exception-Mechanismus behandelt. Für die einfache Verwendung stellt die Field-Klasse statische Factory-Methoden ([16, p.107ff], im Diagramm nach UML-Notation unterstrichen dargestellt) für die zulässigen und sinnvollen Kombinationen bereit. Zu Beachten ist, dass bei Lucene statische Methoden entgegen den Java-Code-Konventionen von Sun ([42]) mit Großbuchstaben beginnen.

Sollen Datumsangaben zur Suche bereitgestellt werden, so müssen diese in ein einheitliches, geeignetes String-Format konvertiert werden. Dies erfolgt durch statische Methoden der Klasse org.apache.lucene.document.DateField.

Die restlichen Methoden der Document- und Field-Klasse dienen dem Zugriff auf die enthaltenen Informationen. Bei Bedarf sei hier auf die Javadoc ([2]) verwiesen.

## *3.5 Indizierung - Konzept*

Die Indizierung dient der Überführung der zuvor erstellten *Dokumente* in eine Index-Datenbank.

Dem Entwickler ist überlassen, verschiedene Optionen beim Indizierungsvorgang zu definieren. Diese betreffen die Ablage des Index (z.B. Dateisystem oder Hauptspeicher), sowie die Definition wie die Texte des Suchraumes analysiert werden sollen.

### 3.5.1 Analyse der Texte

Der Ablauf bei der Analyse der Texte kann grob in zwei Schritte unterteilt werden.

Zunächst müssen die Eingangstexte in Folgen von so genannten ***Tokens*** zerlegt werden. Bei den meisten Texten sind Tokens gleichbedeutend mit Wörtern. Der Mechanismus wird als Tokenizer bezeichnet und ist austauschbar, um auch spezielle Formate zu unterstützen. Zu jedem Token werden der Inhalt und Meta-Informationen erfasst. Die Meta-Informationen

umfassen die Start- und Ende-Position innerhalb des Textes. Wichtig ist, dass diese Zerlegung durchgeführt wird, bevor irgendwelche Bereinigungen erfolgen. Würden vorher Bereinigungen durchgeführt, so könnten die Positionen innerhalb der Texte nicht korrekt ermittelt werden. Die einzelnen Tokens können beliebig typisiert werden. Ein Mechanismus typisiert die Tokens durch Analyse ihrer Struktur. So wird eine Klassifizierung in normale Worte, Worte mit Apostrophen und eine Reihe spezieller Wortformen, wie email-Adressen, Server-Namen, Firmenbezeichnungen (z.B. "AT&T"), Akronyme (z.B. "U.S.U.") und Satzzeichen vorgenommen. Dieser Mechanismus arbeitet auf Basis einer Grammatik analog den in Compilern eingesetzten Parser für Programmiersprachen. Durch den Einsatz eines generierten Parsers ist die Klassifizierung der Tokens effizienter als eine spätere manuelle Analyse.

Anschließend wird die Wortfolge durch Filter aufbereitet. Filter können hierbei Wörter auslassen (z.B. *StopWords*), verändern (z.B. *CaseFolding*) oder hinzufügen (z.B. *Synonym-Suche*). Filter können in beliebiger Anzahl eingesetzt werden.

Abbildung 3-3 zeigt beispielhaft den logischen Ablauf bei der Analyse eines Textes unter Anwendung eines Tokenizers und verschiedener Filter.

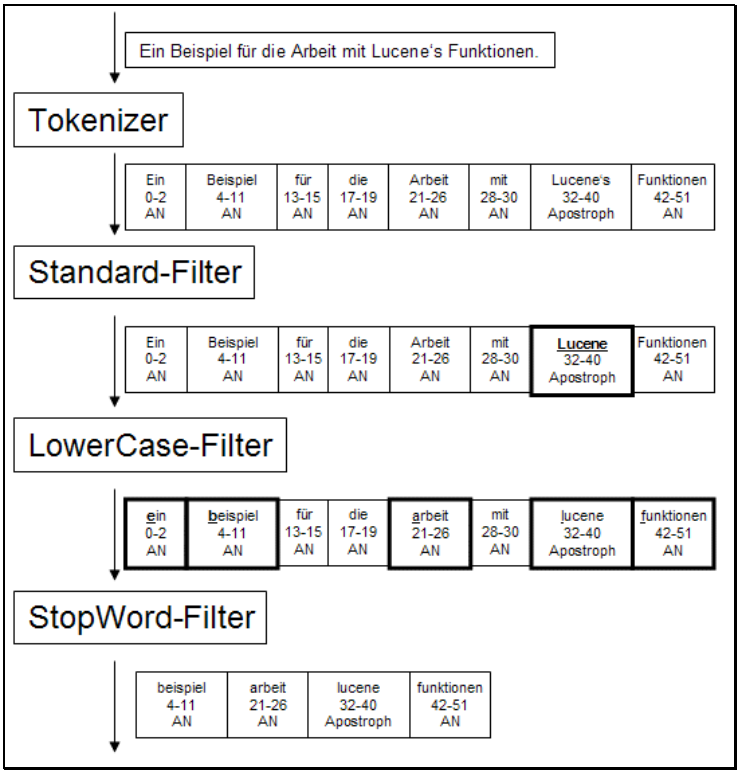


**Abbildung 3-3 Beispiel für die Analyse eines Textes**

Der *Tokenizer* trennt zunächst den Eingangstext in eine Folge von Wörtern. Die Wörter werden klassifiziert in alphanumerisch (AN) und Wörter mit Apostrophen. Durch den *Standard-Filter* werden die Texte normalisiert. Dabei wird die Endung "'s" aus dem Wort "Lucene's" entfernt. Um unnötige Stringvergleiche zu vermeiden nutzt der Filter dabei die vorhandene Klassifizierung. Der *LowerCase-Filter* wandelt alle Großbuchstaben in Kleinbuchstaben um. Anschließend entfernt der *StopWord-Filter* Wörter, welche für die Suche meist nicht relevant sind.

Der Eingangstext "Ein Beispiel für die Arbeit mit Lucene's Funktionen" wird somit reduziert auf die Folge der Wörter "beispiel", "arbeit", "lucene" und "funktionen". Diese Wörter werden im Index gespeichert.

Lucene liefert verschiedene Tokenizer mit. Die Unterschiede zwischen den Tokenizern liegen hauptsächlich im Kriterium für die Trennung einzelner Tokens. Der oben verwendete Grammatik-basierte Tokenizer verwendet Leerzeichen als Trennkriterium und ist der einzige Tokenizer, welcher eine Klassifizierung der Tokens vornimmt. Andere Tokenizer nehmen als Kriterium für die Trennung Ziffern, Leerzeichen oder Nicht-Buchstaben. Manche Tokenizer übernehmen aus Performanzgründen auch gleich die Umwandlung der Tokens in Kleinbuchstaben.

Filter sind bereits vorhanden für *StopWords*, *CaseFolding*, Normalisierung sowie deutsche und englische Wortstammsuche (*Stemming*).

Eine wichtige Eigenschaft von Lucene ist, dass die Anzahl der Wörter, welche in einem Feld indiziert werden, begrenzt ist um den Hauptspeicherbedarf zu begrenzen. Diese Grenze ist standardmäßig bei 10.000 Wörtern, kann jedoch verändert werden (siehe Abschnitt 3.6). Ist ein Text länger als diese Grenze, so wird der Text nur bis zu dieser Grenze analysiert. Der restliche Inhalt wird ignoriert und kann somit für eine Suche nicht verwendet werden!

### 3.5.2 Ablageort für den Index

Lucene liefert Implementierungen für die Ablage des Index im Dateisystem oder im Hauptspeicher. Die Ablage des Index im Hauptspeicher ermöglicht gegenüber der Ablage im Dateisystem eine höhere Performance, da keine teuren Plattenzugriffe benötigt werden. Dies wird jedoch mit dem Nachteil erkauft, dass der Index nicht persistent ist, d.h. bei Programmende verloren geht.

Die Möglichkeiten der Ablage können durch eigene Implementierungen erweitert werden.

Naheliegend wäre eine Ablage des Index in einer Datenbank, da diese performante Zugriffs- und Suchmethoden bereitstellt. Diese können jedoch nicht für die Suche verwendet werden, da die Schnittstelle nur den Zugriff auf Dateien und deren Inhalte regelt. Die Schnittstelle für den Ablageort besitzt keinerlei Semantik über die Struktur der Dateien oder dem Suchzugriff. Die Strukturierung wird durch die Indizierungskomponente vorgenommen und erfordert einen wahlfreien Zugriff. Würde eine Datei durch eine Datenbank abgebildet, so müsste der Inhalt in Form eines BLOB-Feldes abgelegt werden. Bei einem Zugriff auf diese Datei müsste der gesamte Inhalt aus der Datenbank in den Hauptspeicher geladen werden.

Die Indizierungskomponente stellt den Kern von Lucene dar und kann nicht ausgetauscht werden.

### 3.5.3 Speicherung und Aufbau des Index

Die meisten Suchmaschinen verwenden B-Bäume ([38, p. 308]) zur Darstellung der Daten in einem einzelnen Index ([17]). B-Bäume sind in Bezug auf Einfügungen relativ stabil und haben gute I/O-Eigenschaften. Der Nachteil ist eine relativ aufwändige Implementierung, insbesondere um parallele Lese- und Schreibzugriffe zu ermöglichen. Lucene verwendet hier einen alternativen Ansatz und vermeidet so den Einsatz komplexer B-Bäume.

Das Konzept gehört zur Klasse der *Invertierten Indizes* (Abschnitt 2.8.1).

Bei der Indizierung ist der Festplattenzugriff meist ein Engpass, weswegen bei Lucene der Ansatzpunkt für Optimierungen in der Verminderung von Plattenzugriffen liegt.

Ein zentrales Konzept bei Lucene ist die Verwendung von ***Segmenten***. Dabei handelt es sich technisch um vollständige Indizes, welche einzeln durchsucht werden können. Die Dokumente können über mehrere Segmente verteilt sein, bei einer Suche werden dann alle Segmente sequentiell durchsucht und die Ergebnisse zusammengeführt.

Bei der Indizierung werden von Lucene zunächst mehrere kleine Segmente erzeugt und anschließend kombiniert, wie Abbildung 3-4 illustriert.

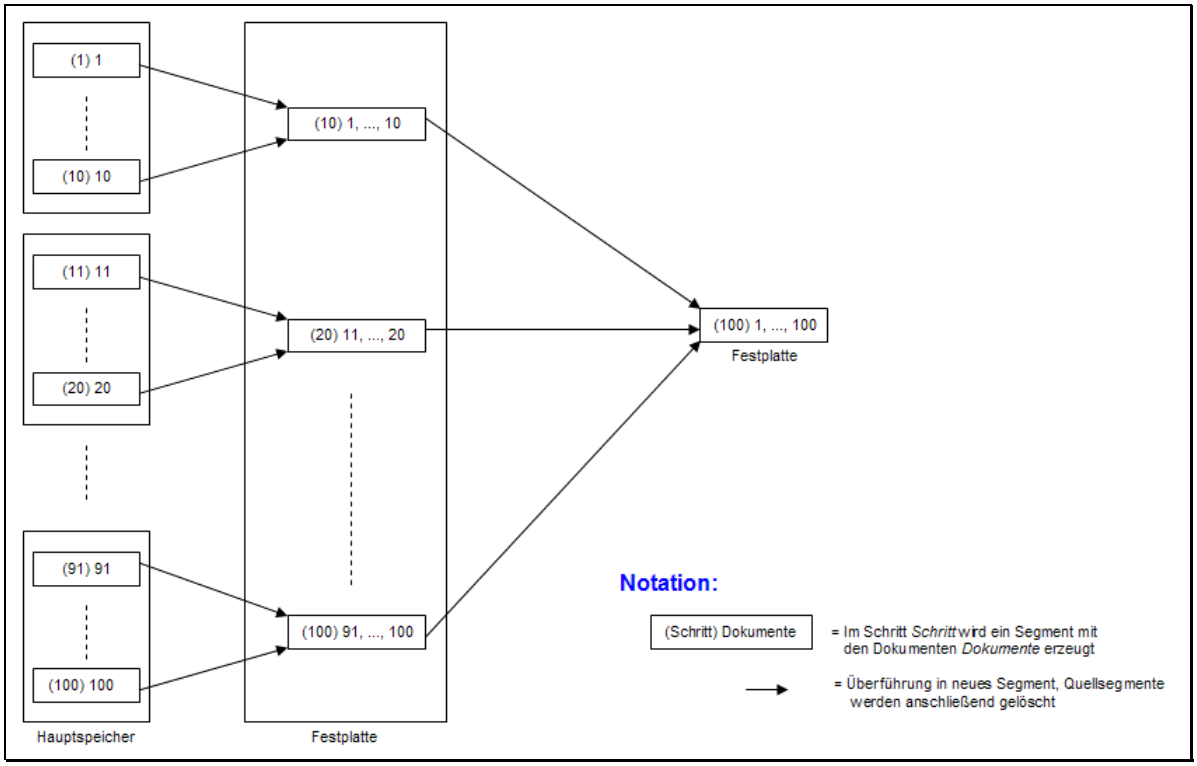


**Abbildung 3-4 Zusammenführung der Segmente**

Für jedes neue indizierte Document-Objekt wird zunächst ein neues Segment im Hauptspeicher angelegt. Sind im Hauptspeicher 10 (eingestellt durch die öffentliche Variable `mergeFactor`) Segmente vorhanden, so werden diese in ein einzelnes neues Segment mit 10 Einträgen auf der Festplatte zusammengeführt. Sind auf der Festplatte 10 dieser Segmente vorhanden, so werden diese wiederum in ein neues Segment mit 100 Einträgen zusammgeführt. Dies wird solange durchgeführt bis die Segmentgröße den Wert der öffentlichen Variable `maxMergeDocs` erreicht. Diese Variable ist voreingestellt auf den Wert Integer.MAX_VALUE (2.147.483.647) und wird in der Realität nur in Einzelfällen erreicht werden.

Optimierungen bezüglich der Indizierungs-Performance können also insbesondere durch Änderung der Konstante `mergeFactor` durchgeführt werden. Ein höherer Wert beschleunigt die Indizierung, benötigt aber mehr Hauptspeicher. In [19] werden die Auswirkungen dieses Faktors untersucht.

Gleichzeitige Lese- und Schreibzugriffe auf einen Index können Konflikte verursachen. Diese Konflikte können über komplizierte Locking-Mechanismen behoben werden. Lucene setzt hier alternativ ein einfaches aber effektives Verfahren ein: Bestehende Segmente werden niemals geändert, so dass hier Lesezugriffe nicht gestört werden. Alle Änderungen werden nur in neuen Segmenten durchgeführt. Werden mehrere Segmente zusammengefügt, so wird ein neues Segment erzeugt. Die alten Segmente werden gelöscht, sobald die letzte vorher begonnene Suchanfrage bearbeitet wurde.

Lucene bietet die Möglichkeit einer Index-Optimierung. Dabei werden alle vorhandenen Segmente in ein einziges Segment zusammengeführt. Dies optimiert den Index für schnelle Suchen, da nicht mehrere Segmente durchsucht werden müssen. Gleichzeitig verringert es durch die eingesetzten Komprimierungstechniken die Gesamtgröße. Insbesondere Anwendungen welche nur unregelmäßigen Datenänderungen unterworfen sind, können auf diese Weise auf die Suche weiter optimiert werden.

Das Indizierungsverfahren hat gute Skalierungseigenschaften und bietet dem Entwickler ein hohes Maß an Flexibilität zum Ausgleich zwischen Indizierungs- und Suchgeschwindigkeit. Gleichzeitig hat das Verfahren gute I/O-Eigenschaften sowohl bei der Indizierung als auch bei der Suche ([17]). Der Nachteil ist ein größerer temporär benötigter Platzbedarf beim Zusammenführen. Durch die Erzeugung neuer Segmente wird hier der bereits von den Segmenten eingenommene Platz nochmals temporär benötigt. Aufgrund der heute verfügbaren Festplattengrößen stellt dies jedoch nur bei extrem großen Indizes ein Problem dar.

Da Segmente nicht aktualisiert werden sind hier keine komplizierten B-Bäume notwendig. Schnelle Suchen werden durch eine Reihe von Komprimierungstechniken (siehe Abschnitt 3.6.4), welche Plattenzugriffe vermindern ohne extreme CPU-Belastung zu erzeugen, ermöglicht. Zusätzlich wird ein Teil des Index bei der Suche im Hauptspeicher gehalten. Durch diese Techniken wird eine Komprimierung auf bis zu 30% der ursprünglichen Textgröße erreicht ([37]).

### 3.5.4 Dateien des Index

Für die Lucene-Version 1.2 existiert keine Beschreibung der Dateiformate. Aus diesem Grund basieren die Erläuterungen dieses Abschnitts auf der Beschreibung der Formate in der Version 1.3 ([4]). Die Version 1.3 ist als Release-Candidate 1 seit dem 20. März 2003 verfügbar.

Ein Index wird jeweils in einem Verzeichnis abgelegt. Die dabei erstellten Dateien können in zwei Gruppen unterteilt werden.

Indexspezifische Dateien besitzen einen festen Namen und existieren einmal pro Index.

Segmentspezifische Dateien existieren einmal pro Segment. Alle Dateien eines Segments haben identische Dateinamen (den Segmentnamen) und unterscheiden sich nur durch deren Extensionen.

Abbildung 3-5 bietet eine Übersicht über die verwendeten Dateien. Tabelle 3-2 beschreibt die Dateien, welche einmalig pro Index vorhanden sind, Tabelle 3-3 beschreibt die Segment-spezifischen Dateien.

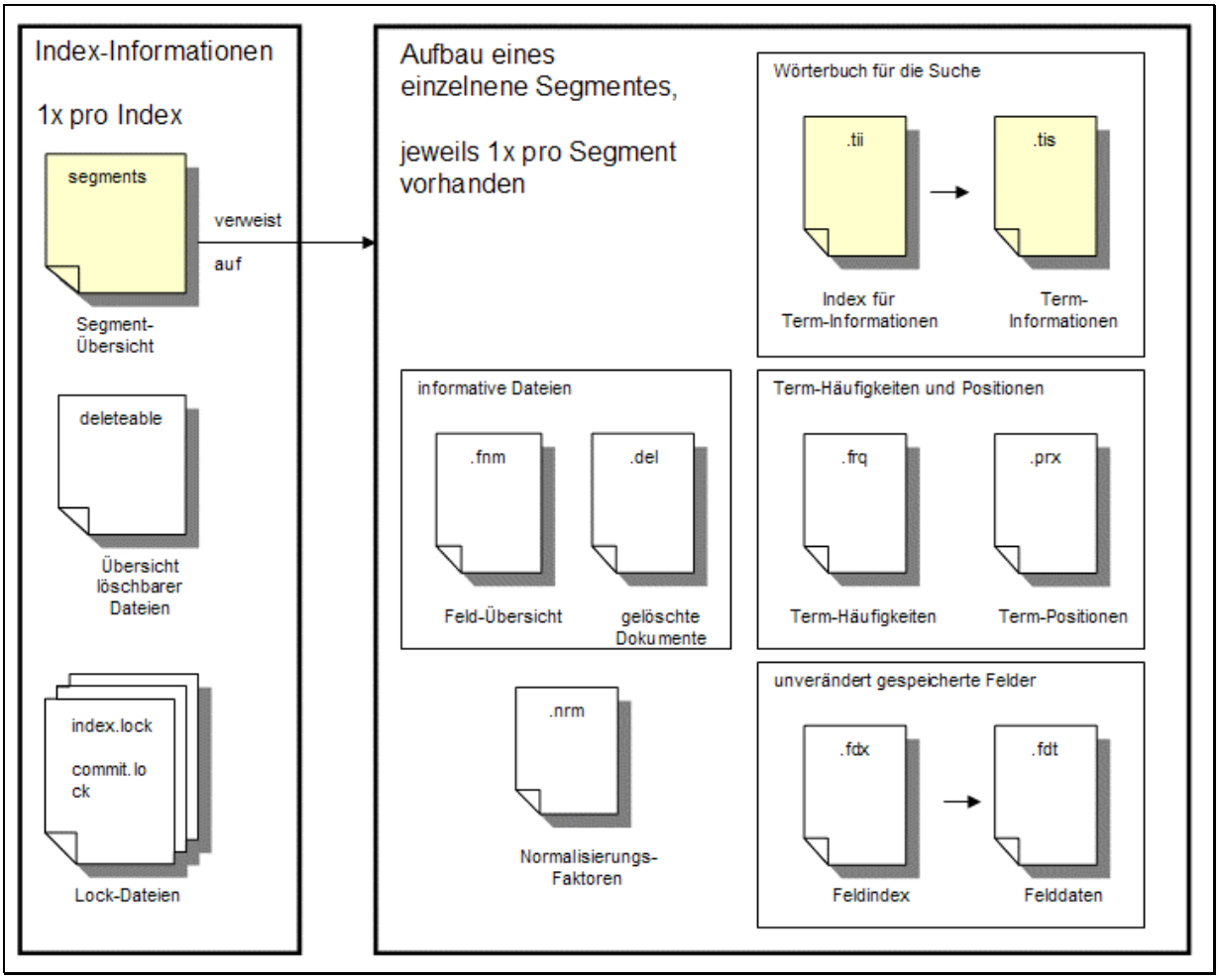


**Abbildung 3-5 Übersicht Dateien eines Index**

| Datei | Inhalt und Zweck |
|---|---|
| Segment-Übersicht (segments) | • bietet Informationen über die aktiven Segmente des Index (Anzahl, sowie deren Namen und Größe. |
| Übersicht löschbarer Dateien (deleteable) | • listet alle Dateien auf, welche nicht länger benutzt werden, aber nicht gelöscht werden konnten. Die Datei ist nur auf Windows-Systemen vorhanden, da Windows keine geöffneten Dateien löschen kann. |
| Lock-Dateien (index.lock, commit.lock) | • "index.lock" kennzeichnet, dass ein Prozess gegenwärtig Dokumente in den Index einfügt. Dies verhindert parallele Schreibzugriffe.<br>• "commit.lock" kennzeichnet, dass ein Prozess gerade die Segment-Informations-Datei bearbeitet oder liest. Die Datei verhindert, dass ein Prozess Segmente löscht, während ein anderer Prozess die Segment-Informationen gelesen hat, aber noch nicht alle Dateien öffnen konnte. |

**Tabelle 3-2 Dateien für die Informationen des Index**

| Datei | Inhalt und Zweck |
|---|---|
| Feld-Übersicht (*.fnm) | • listet alle Felder auf, zusammen mit der Information ob diese indiziert wurden.<br>• weist Feldern interne Nummern zu. |
| gelöschte Dokumente (*.del) | • listet Dokument auf, welche gelöscht wurden aber noch im Index vorhanden sind (bei der Suche werden diese ignoriert) |
| unverändert gespeicherte Felder (*.fdx, *.fdt) | • *Felddaten* speichert die Feld-Inhalte für Felder, wie Schlüsselwörter oder Referenzen<br>• Der *Feldindex* dient dem schnellen Auffinden der Felder eines speziellen Dokuments. Die Datei ist für wahlfreien Zugriff optimiert und speichert die Anfangspositionen der Dokumente in der Felddaten-Datei |
| Wörterbuch (*.tii, *.tis) | • Kern des Index (s.u.) |
| Term-Häufigkeiten (*.frq) | • enthält für jedes Dokument und jeden darin enthaltenen Term dessen Häufigkeit im jeweiligen Dokument.<br>• wird zur Berechnung der Relevanz benötigt. |
| Term-Positionen (*.prx) | • enthält die Positionen der Vorkommnisse jedes Terms in jedem Dokument.<br>• dient der Auffindung der Textstellen bei der Suchausgabe. |
| Normalisierungsfaktoren (*.nrm) | • enthält zu jedem Dokument einen Fließkomma-Faktor, welcher bei der Berechnung der Relevanz als Multiplikant verwendet wird. Dadurch können bestimmte Dokumente höher bewertet werden als andere.<br>• neues Feature der Version 1.3. Der Wert wird durch eine neue Methode der Klasse Document eingestellt. |

**Tabelle 3-3 Dateien der Segmente**

Das Wörterbuch für die Suche ist der Kern des Index. Die Datei "Term-Informationen" baut einen *Invertierten Index* (Abschnitt 2.8.1) auf. Die Datei enthält alle Terme verknüpft mit den Feldern und der Anzahl der Dokumente, welche diesen enthalten. Zusätzlich enthält sie Verweise auf die Dateien "Term-Häufigkeiten" und "Term-Positionen". Der "Index für die Term-Informationen" enthält prinzipiell die selben Informationen, allerdings nur für jeden 128en Term zusammen mit einem Verweis auf die Position in den Term-Informationen. Diese Datei wird in den Hauptspeicher geladen, so dass der Großteil der Suche ohne Plattenzugriffe erfolgen kann. Wurde über diese Datei der Bereich gefunden, so erfolgt die restliche Suche dann in den Term-Informationen.

Der Hauptunterschied zur Version 1.2 liegt darin, dass alle Felder in einer einzelnen Datei abgelegt werden. Die Version 1.2 verwendet pro Feld und Segment eine Datei. Die Gesamtmenge an Dateien ist hier also wesentlich größer. Durch die Verminderung der Anzahl der Dateien ist in der kommenden Version 1.3 eine Performanzsteigerung zu erwarten, da die Anzahl der nötigen I/O-Zugriffe vermindert werden.

## 3.6 Indizierung - Implementierung

Einfüge-Operationen werden durch die Klasse org.apache.lucene.index.IndexWriter gekapselt. Diese wird ausführlich in Abschnitt 3.6.3 beschrieben.

Löschungen erfolgen über die Klasse org.apache.lucene.index.IndexReader, welche primär für die Suche zuständig ist. Der Grund warum dies durch den IndexReader und nicht durch den IndexWriter erfolgt ist nicht dokumentiert. Wahrscheinlich liegt dies daran, dass die zu löschenden Dokumente auch über einen Suchmechanismus ermittelt werden. Zum Löschen eines Dokumentes muss entweder die interne Dokument-Nummer oder der genaue Inhalt eines Keyword-Feldes bekannt sein. Die interne Dokument-Nummer ist jedoch nicht konstant sondern kann sich durch Einfügungen oder Löschungen im Index ändern. Die Dokument-Nummer kann auch durch eine Suche ermittelt werden.

Die Definition der vorhandenen Parameter erfolgt über den Konstruktor der IndexWriter-Klasse. Dieser erwartet drei Parameter. Der erste Parameter definiert den Ablageort für den Index. Der zweite Parameter definiert wie die Daten analysiert werden. Der dritte Parameter definiert, ob ein neuer Index angelegt, oder ein bestehender Index erweitert werden soll.

### 3.6.1 Analyse der Texte

Zugangspunkt für die Analyse der Texte ist eine Implementierung der abstrakten Klasse org.apache.lucene.analysis.Analyzer. Diese Klasse folgt dem Abstract-Factory-Pattern ([16, p. 87ff]) und ist in Abbildung 3-6 dargestellt.

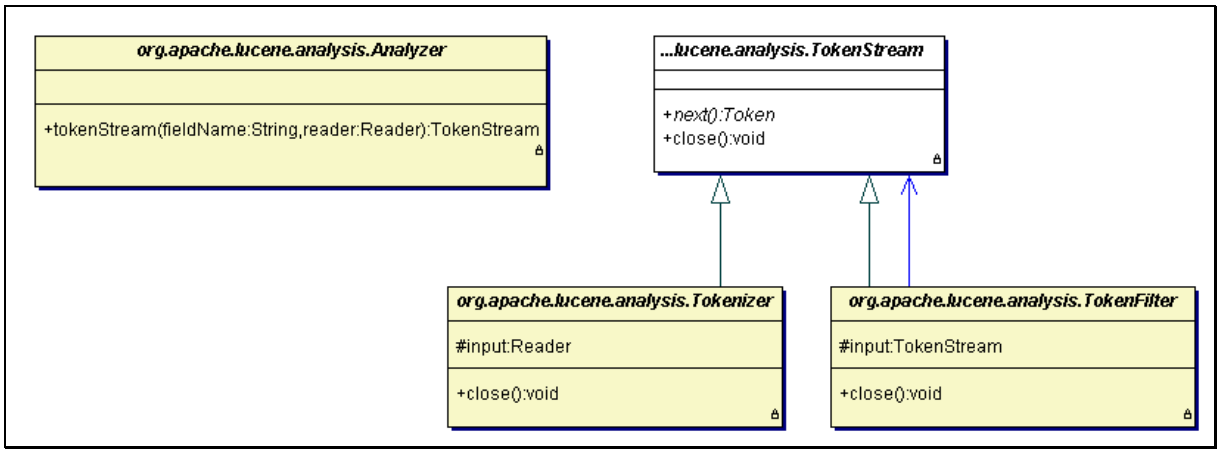


**Abbildung 3-6 Basisklassen für Text-Analyse**

Ein Analyzer erzeugt über die Factory-Methode `tokenStream` ein Tokenizer- und bei Bedarf mehrere TokenFilter-Objekte.

Die Aufgabe eines Tokenizers ist die Umwandlung der von einem Reader erzeugten Folge von Zeichen in Tokens.

TokenFilter können nun diese Tokens weiterverarbeiten. Dabei können auch Transformationen vorgenommen werden, wie beispielsweise die Umwandlung von Groß- in Kleinbuchstaben. Prinzipiell können Filter auch neue Tokens einfügen, müssen dann allerdings intern einen Puffer implementieren.

Eine Reihe von Tokenizern sind im Package org.apache.lucene.analysis vordefiniert. Tabelle 3-4 stellt diese zusammen.

| Klasse | Kriterium für die Trennung zweier Tokens |
|---|---|
| LetterTokenizer | nimmt Trennung an Zeichen vor, welche keine Buchstaben sind. |
| LowerCaseTokenizer | LetterTokenizer, welcher Wörter in Kleinbuchstaben umwandelt. Dies ist performanter als die spätere Anwendung eines Filters. |
| WhitespaceTokenizer | nimmt Trennung an Leerzeichen vor. |
| StandardTokenizer | nimmt Trennung auf Basis einer Grammatik vor und klassifiziert die Tokens. Die Grammmatik ist laut JavaDoc für die meisten europäischen Sprachen geeignet. |

**Tabelle 3-4 Vordefinierte Tokenizer**

Zusätzlich zu den Tokenizern ist auch eine Reihe von TokenFiltern vordefiniert. Diese sind ebenfalls im Package org.apache.lucene.analysis definiert. Sprachabhängige Filter befinden sich in entsprechenden Unterpackages. Diese stellt Tabelle 3-5 zusammen.

| Klasse | Arbeitsweise |
|---|---|
| GermanStemFilter | ersetzt Tokens durch den deutschen Wortstamm (Algorithmus basiert auf [8], siehe Abschnitt 2.6.2). |
| LowerCaseFilter | wandelt alle Großbuchstaben in Kleinbuchstaben um. |
| PorterStemFilter | ersetzt Tokens durch den englischen Wortstamm (Algorithmus in [33], siehe Abschnitt 2.6.2). |
| StandardFilter | entfernt Endungen wie " 's", entfernt Punkte aus Akronymen, nutzt dabei Klassifizierung des StandardTokenizers. |
| StopFilter | entfernt StopWords, welche im Konstruktor angegeben werden. |

**Tabelle 3-5 Vordefinierte TokenFilter**

Die vorhandenen Tokenizer und TokenFilter werden durch vordefinierte Analyzer zusammengefügt. Diese sind im Package org.apache.lucene.analysis bzw. sprachabhängig in einem Unterpackage definiert. Tabelle 3-6 stellt diese zusammen.

| Klasse | Tokenizer und TokenFilter |
|---|---|
| GermanAnalyzer | StandardTokenizer, StandardFilter, StopFilter (deutsch als Standard, alternative Wortliste möglich), GermanStemFilter (variable Exclude-Liste), LowerCaseFilter |
| SimpleAnalyzer | LowerCaseTokenizer |
| StandardAnalyzer | StandardTokenizer, StandardFilter, LowerCaseFilter, StopFilter (englisch als Standard, alternative Wortliste möglich) |
| StopAnalyzer | LowerCaseTokenizer, StopFilter (englisch als Standard, alternative Wortliste möglich) |
| WhitespaceAnalyzer | WhitespaceTokenizer |

**Tabelle 3-6 Vordefinierte Analyzer**

Die typische Implementierung der Factory-Methode Analyzer.`tokenStream` ist in Code-Snippet 3-1 dargestellt.

```
1: public TokenStream tokenStream(String pFieldName, Reader pReader) {

2:   TokenStream tResult = new AnyTokenizer(pReader);

3:   tResult = new TokenFilter1(tResult);
4:   tResult = new TokenFilter2(tResult);
5:   tResult = new TokenFilter3(tResult);

6:   return tResult;
7: }
```

**Code-Snippet 3-1 Implementierung eines Analyzers**

In Zeile 2 wird ein Tokenizer instanziiert, welcher aus dem übergebenen Reader-Objekt pReader ein TokenStream-Objekt erzeugt. Dieses TokenStream-Objekt wird nun in den Zeilen 3-5 wiederholt einem TokenFilter-Objekt übergeben. Das TokenFilter-Objekt implementiert ebenfalls einen TokenStream und kann daher als Ersatz weitergegeben werden (siehe Abbildung 3-6).

Die Reihenfolge der Filter kann relevant sein. Einige Filter stellen Anforderungen an die Eingangsdaten. So verlangt der PorterStemFilter beispielsweise, dass die Tokens bereits in Kleinbuchstaben umgewandelt wurden. Für die Anforderungen der einzelnen Filter sei auf die JavaDoc ([2]) verwiesen.

Aus Performanzgründen sollten Filter, welche eine Vielzahl von Tokens auslassen (z.B. StopFilter) möglichst früh definiert werden, um die Datenmenge für spätere Filter zu begrenzen.

Durch den Parameter `pFieldName` kann die Analyse feldabhängig variiert werden, was im Beispiel jedoch nicht erfolgt.

### 3.6.2 Ablageort für den Index

Die Definition des Ablageortes erfolgt über eine Instanz einer Implementierung der abstrakten Klasse org.apache.lucene.store.Directory. Wie bereits in Abschnitt 3.5.2 angesprochen existieren Implementierungen für die Ablage im Dateisystem (FSDirectory) und im Hauptspeicher (RAMDirectory).

Abbildung 3-7 zeigt diese Schnittstelle.

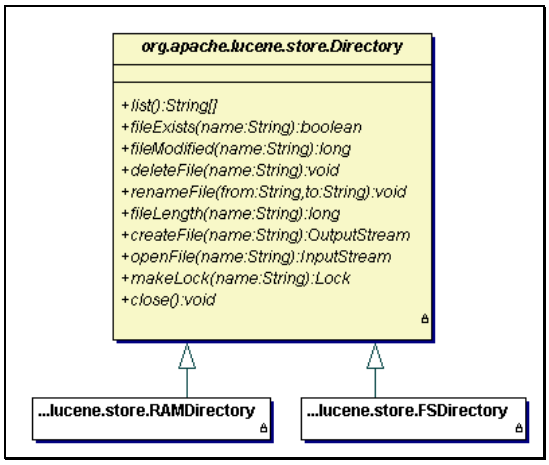


**Abbildung 3-7 Implementierung verschiedener Ablageorte**

Ein Directory entspricht logisch einem Verzeichnis auf einer Festplatte mit eindeutig benannten Dateien als Inhalt. Implementierungen von Directory müssen wie in Abbildung 3-7 dargestellt eine Reihe von Methoden bereitstellen.

Die Funktionen der meisten Methoden sind hier nicht relevant und intuitiv am Namen erkennbar. Für eine genaue Beschreibung dieser Methoden sei hier auf die JavaDoc ([2]) verwiesen.

Die Methoden `createFile` und `openFile` dienen dem Schreib- bzw. Lesezugriff auf den Inhalt einer Datei. Wie in Abbildung 3-7 zu sehen ist, arbeitet dieser Zugriff ohne Kenntnis der inneren Struktur lediglich auf Streams. Insbesondere werden keinerlei Informationen bezüglich der Indexstruktur oder dessen Inhalt übergeben. Dadurch wird deutlich, wieso Datenbank-Such-Funktionalitäten nicht verwendet werden können.

Durch die Methode `makeLock` wird ein Lock mit dem angegebenen Namen erzeugt. Dieses kennzeichnet, dass der Index momentan beschrieben wird und verhindert konkurrierende Schreib-Zugriffe. Beim Öffnen eines Index über einen IndexWriter und beim Versuch ein Objekt über den IndexReader zu löschen wird hierzu die Existenz eines Locks geprüft und gegebenenfalls der Versuch mit einer Exception quittiert.

Dem Konstruktor der IndexWriter-Klasse kann im ersten Parameter ein Directory-Objekt, ein java.io.File-Objekt oder ein String-Objekt übergeben werden. Wird ein File oder String übergeben, so wird intern ein FSDirectory-Objekt erzeugt und das File- bzw. String-Objekt als Verzeichnis für den Index verwendet.

Die Schnittstelle bietet den Vorteil der einfachen Implementierung verschiedener *Directories*, da deren Funktionalität weitgehend unabhängig von Lucene ist. Gleichzeitig verhindert sie aber die Nutzung evtl. vorhandener Such-Funktionalitäten im Ablagesystem.

### 3.6.3 Speicherung und Aufbau des Index

Die Speicherung ist Aufgabe der Klasse IndexWriter, welcher in Abbildung 3-8 dargestellt ist.

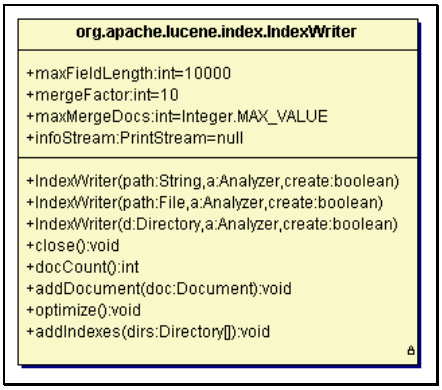


**Abbildung 3-8 Zugriff auf den Index über die Klasse IndexWriter**

Die Variable `maxFieldLength` definiert die bereits in Abschnitt 3.5.1 angesprochene Begrenzung der Wörter, welche in einem Feld indiziert werden. Durch die Variablen `mergeFactor` und `maxMergeDocs` kann die in Abschnitt 3.5.3 beschriebene Segment-Zusammmenführung parameterisiert werden. Der Variablen `infoStream` kann ein PrintStream-Objekt zugewiesen werden. Dieser PrintStream wird über die Aktivitäten bei der Segment-Zusammenführung (Merging) benachrichtigt. Dies dient hauptsächlich der Fehleranalyse oder der Entwicklung von Lucene-Komponenten.

Die drei Konstruktoren wurden bereits in der Einleitung von Abschnitt 3.6 beschrieben. Über die Methode `addDocument` werden neue Dokumente in den Index eingefügt, `docCount()` gibt die Anzahl der momentan im Index vorhandenen Dokumente zurück. Über die Methode `addIndexes` können externe Indizes eingefügt werden. Dies kann verwendet werden, um eine Indizierung in mehreren parallelen Prozessen durchzuführen. Durch diese Methode werden die übergebenen Indizes in den Index eingefügt und anschließend der Index optimiert. Die Methode `optimize` optimiert den Index, wie in Abschnitt 3.5.3 beschrieben.

Sind die Änderungen erfolgt, so wird der Index über die Methode close() geschlossen und noch im Hauptspeicher vorhandene Segmente werden auf Festplatte geschrieben.

### 3.6.4 Format eines Segmentes

Wie Abschnitt 3.5.4 baut auch dieser Abschnitt auf der Beschreibung der Dateiformate in der Lucene-Version 1.3 auf. Der genaue Aufbau der einzelnen Dateien ist in [4] beschrieben. An dieser Stelle sollen nur die wichtigsten Komprimierungstechniken beschrieben werden.

Zahlen, welche sehr große Bereiche benötigen, werden in variablen Längen gespeichert (***VInt***). Hierzu werden von einem Byte für die Zahl nur sieben Bit verwendet. Das höchstwertigste achte Bit definiert, ob ein weiteres Byte zur Speicherung der Zahl verwendet

wird. Ist dies der Fall, so werden dessen sieben niedrigwertige Bit als höherwertige Bits an die Zahl angefügt. Dieses Verfahren kann prinzipiell beliebig oft angewendet werden. Die inhärente Kompression führt dazu, dass Zahlen bis 127 in einem Byte, Zahlen von 128 bis 16383 in zwei Bytes, Zahlen von 16384 bis 2097151 in drei Bytes usw. gespeichert werden. Die Codierung ist in Tabelle 3-7 illustriert.

| Wert | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|
| 0 | 0 0000000 | - | - |
| 1 | 0 0000001 | - | - |
| ... | | | |
| 127 | 0 1111111 | - | - |
| 128 | 1 0000000 | 0 0000001 | - |
| 129 | 1 0000001 | 0 0000001 | - |
| ... | | | |
| 16.383 | 1 1111111 | 0 1111111 | - |
| 16.384 | 1 0000000 | 1 0000000 | 0 0000001 |
| 16.385 | 1 0000001 | 1 0000000 | 0 0000001 |
| ... | | | |

**Tabelle 3-7 Beispiel für die Codierung von Integers variabler Länge**

Diese Codierung bietet eine Kompression und ist dennoch effizient zu decodieren. Das Verfahren benötigt in Einzelfällen (wie beispielsweise die Zahlen 128-255) allerdings mehr Platz als eine Standard-Byte-Codierung, verhindert dafür aber eine Beschränkung des Wertebereiches ([4]).

Im Wörterbuch werden die einzelnen Terme alphabetisch sortiert abgelegt. Eine Komprimierung erfolgt, indem der Beginn aus dem vorigen Term mitbenutzt wird. Um den vollständigen Term zu erhalten werden aus dem vorigen Term die ersten *Präfix-Länge* Zeichen benutzt und um das gespeicherte *Suffix* (String) erweitert. Tabelle 3-8 illustriert dies an einem Beispiel.

| Eintrag-Nr | Term | Präfix-Länge | Suffix |
|---|---|---|---|
| 1 | bahnhof | 0 | bahnhof |
| 2 | bauhaus | 2 | uhaus |
| 3 | baumarkt | 3 | markt |
| 4 | baumaschine | 5 | schine |
| 5 | baumaschinen | 11 | n |

**Tabelle 3-8 Beispiel für die Kompression im Wörterbuch**

Unter der Voraussetzung, dass keine zusätzlichen Informationen abgelegt werden, und jedes Zeichen jeweils ein Byte Platz benötigt, würde ohne Komprimierung ein Platzbedarf von 7+7+8+11+12 = 45 Byte benötigt. Mit Hilfe dieser Komprimierung wird dies im Beispiel bereits auf 7+5+5+6+1 (+5) = 29 Byte verringert. Die fünf Byte sind für die Speicherung der

Präfix-Länge erforderlich. Der Platzbedarf wurde in diesem einfachen Beispiel bereits auf 64,4% verringert. Mit steigender Indexgröße ist hier auch eine weitere Verringerung zu erwarten, da die Wahrscheinlichkeit identischer Term-Präfixe steigt.

Der Nachteil dieser Variante ist, dass die Datei zwingend sequentiell gelesen werden muss. Die sequentielle Analyse der gesamten Datei benötigt allerdings viel Zeit und I/O-Zugriffe. Aus diesem Grund wird jeder 128e Eintrag in den Term-Info-Index eingetragen. Dabei werden die Inhalte der Felder *Präfix-Länge* und *Suffix* entsprechend angepasst. Der Sinn der zusätzlichen Index-Datei ist es, einen fast wahlfreien Zugriff auf die Term-Info-Datei zu erhalten. Die Datei ist dazu konzipiert, dass sie komplett in den Hauptspeicher geladen wird, so dass I/O-Zugriffe erst dann erfolgen müssen, wenn die Anzahl der zu analysierenden Terme auf 127 begrenzt wurde.

Durch die Verwendung von 32-Bit-Integer Werten in verschiedenen Dateien wird die maximale Anzahl von Dokumenten und Termen auf 4,3 Millarden begrenzt. Diese Begrenzung ist heute noch unproblematisch. Aufgrund der ständig wachsenden Datenmengen kann dies in Zukunft jedoch zu einem Problem werden. Die Entwickler haben dies erkannt und bereits vorgeschlagen, die Werte in ein UInt64 oder noch besser einen *VInt* zu verwandeln, so dass hier dann keine Beschränkungen mehr vorhanden sind.

## *3.7 Anfragebearbeitung - Konzept*

Die Anfragebearbeitung dient der Definition einer Suchanfrage und bei Bedarf deren Überführung in ein für die anschließende Suche geeignetes Format. Die Suche erwartet die Anfrage in Form eines Objektmodells (***Objekt-Anfrage***).

Dieses Objektmodell kann durch Instanziierung und Komposition direkt erzeugt werden. Zusätzlich bietet Lucene die Möglichkeit, die Anfrage als String (***String-Anfrage***) zu definieren.

Bei der String-Anfrage erfolgt die Bearbeitung in zwei Schritten. Zunächst wird die Anfrage durch einen Parser analysiert und anschließend in eine Objekt-Anfrage überführt. Die String-Variante eignet sich vor allem für allgemeine Suchmaschinen, bei welchen die Kriterien für eine Suche seitens des Benutzers nicht bekannt sind.

Die Definition einer Objekt-Anfrage eignet sich insbesondere dann, wenn die Zusammensetzung der Anfrage bekannt ist, und die Suche durch eine spezialisierte Oberfläche (Wizard / Assistent) unterstützt werden soll. Der Aufbau der Objekt-Anfragen wird in Abschnitt 3.8 beschrieben.

Da bei beiden Verfahren ein Objektmodell derselben Struktur erzeugt wird, weisen beide prinzipiell dieselben Fähigkeiten auf. Die Definition als String kann allerdings abhängig von der Mächtigkeit des Parsers Einschränkungen unterliegen.

Zu beachten ist, dass die Anfrage ebenso aufbereitet werden muss, wie die Texte bei der Indizierung. Wird dies nicht durchgeführt, oder wird hier ein anderes Verfahren verwendet, so kann es passieren, dass die Suchanfrage nicht zu den im Index enthaltenen Daten passt. Wurde beispielsweise bei der Indizierung eine Umwandlung in Kleinbuchstaben vorgenommen, bei der Aufbereitung der Suche jedoch nicht, so führt eine Anfrage nach dem Wort "Haus" zu keinen Ergebnissen, da der Term im Index als "haus" abgelegt ist.

Bei der Analyse der String-Anfrage wird hierzu der Analyse-Algorithmus angegeben. Bei der Objekt-Anfrage erfolgt keine Analyse und Anpassung. Es muss daher durch die Anwendung sichergestellt werden, dass die Texte entsprechend aufbereitet sind. Hierzu kann auch der Analyse-Algorithmus verwendet werden.

### 3.7.1 Elemente einer Suchanfrage

Eine Suchanfrage besteht aus einer oder mehreren Teil-Anfragen, welche durch boolesche Operatoren verknüpft werden. Dabei werden die Suchmodelle *Boolean* (Abschnitt 2.4.1) und *Extended Boolean* (Abschnitt 2.4.3) unterstützt. Komplexe boolesche Verknüpfungen können durch Klammerung geschachtelt werden.

Um Kombinationen beider Suchmodelle zu unterstützen nimmt Lucene eine Erweiterung der Verknüpfungsoperatoren um ein Prädikat vor, welche als "PREFERABLE" bezeichnet werden könnte. Dieser Operator hat keine Auswirkungen auf die Auswahl der Dokumente, sondern nur gemäß des *Extended Boolean* Modells auf die Berechnung der Relevanz. Wird eine Anfrage "Lucene AND PREFERABLE Index" definiert, so werden alle Dokumente gefunden, welche das Wort "Lucene" enthalten. Dokumente, welche zusätzlich das Wort "Index" enthalten, werden aber höher bewertet und erscheinen so weiter oben im Suchergebnis. Um diesen zusätzlichen Operator abzubilden, werden die Bedingungen nicht über die Schlüsselwörter "AND" und "OR" verknüpft, sondern die einzelnen Teilanfragen werden mit den Optionen "erforderlich" und "nicht enthalten" versehen. Dies führt zu vier verschiedenen Verknüpfungstypen. Tabelle 3-9 stellt zeigt deren Bedeutung, wenn zu einer Teilanfrage *x* eine weitere Anfrage *y* mit den angegebenen Parametern angefügt wird.

| Nr | y erforderlich | y nicht enthalten | boolesche Bedeutung |
|---|---|---|---|
| 1 | ja | ja | nicht erlaubt! |
| 2 | ja | nein | AND *y* |
| 3 | nein | ja | AND NOT *y* |
| 4 | nein | nein | PREFERABLE *y* |

**Tabelle 3-9 Parameter boolescher Verknüpfungen**

Der Verknüpfungstyp 1 kann offensichtlich nie erfüllt werden, da er eine Kontradiktion darstellt.

Ergibt eine Teilanfrage keine Ergebnisse, so wird diese als boolesch "falsch" angesehen.

Zu beachten ist, dass reine Ausschlussanfragen (z.B. "Alle Dokumente welche Wort 'x' nicht enthalten") nicht zulässig sind. Diese führen immer zu einem leeren Suchergebnis.

Lucene erlaubt die Verwendung verschiedener Such-Funktionen. Um diese abzubilden, werden die Anfragen typisiert. Jede Anfrage besteht aus einem oder mehreren Termen. Ein Term besteht aus einem Feldnamen und einem String. Wie dieser String interpretiert wird, hängt vom Anfragetyp ab. Durch die Verknüpfung mehrerer Anfragen können die verschiedenen Anfragetypen auch kombiniert werden.

Lucene kennt die Anfragetypen *Term*, *Phrasen*, *Präfix*, *Wildcard*, *Intervall* und *Fuzzy* (vgl. Abschnitt 2.7).

Jede Anfrage (auch die nach einem einzelnen Term) kann zusätzlich mit einem Faktor versehen werden. Dieser wird bei der Berechnung der Relevanz verwendet. Dadurch können einzelne Teilanfragen höher gewichtet werden.

### 3.7.2 String-Anfragen

Die Definition einer String-Anfrage erfolgt analog vieler bekannter Internet-Suchmaschinen (z.B. Google [18]). Die Anfrage wird als String in einem vordefinierten Format gestellt. Für das Format existieren keine allgemein anerkannten Standards, ebensowenig für die Mächtigkeiten der Suchfunktionen. Jede Suchmaschine muss also entsprechend analysiert werden. Für den Benutzer muss in jedem Fall eine Hilfestellung gegeben werden. Die von Lucene vordefinierte Syntax kann bei Bedarf an spezielle Bedürfnisse angepasst werden.

Lucene liefert einen Parser mit, welcher durch den Parser-Generator JavaCC ([27]) generiert wurde. Die zugehörige Grammatik ist als JavaCC-Quelle mitgeliefert, so dass Änderungen der Syntax relativ einfach möglich sind. Eine Einführung zu JavaCC ist in [11] zu finden. Der mitgelieferte Parser unterstützt alle gängigen Suchanfragen. Die genaue Syntax für die String-Suche unter Verwendung des mitgelieferten Parsers ist in [5] beschrieben.

Prinzipiell muss in String-Anfragen jeder einzelne Term vollständig durch Angabe des Feldnamens und des Wertes angegeben werden. Dem Parser wird jedoch beim Aufruf ein Feldname mitgegeben. Dieser Feldname (Default-Feld) wird an den Stellen als Feldname eingesetzt, an denen in der Anfrage kein Name angegeben wurde. Bei der Konzeption einer Suche sollte ermittelt werden, welches Feld in den meisten Fällen für die Suche verwendet wird und als Default-Feld definiert werden.

In Einzelfällen müssen Anfragen als String allerdings sehr komplex gestellt werden. Bei Bedarf kann hier dann auch eine Kombination mit einer Objekt-Anfrage vorgenommen werden. Dabei wird ein Teil der Anfrage als String und der Rest als Objekt-Anfrage formuliert. Der Parser wandelt die String-Anfrage in eine Objekt-Anfrage um. Anschließend werden die beiden Objekt-Anfragen zu einer einzelnen Objekt-Anfrage kombiniert. Dies kann auch mehrfach erfolgen.

Boolesche Verknüpfungen können auf zwei Arten angegeben werden. Entweder werden die allgemein üblichen Operatoren "AND", "OR" und "NOT" verwendet oder die Präfixe "+" (erforderlich) und "-" (nicht enthalten). Der Operator "AND PREFERABLE" kann nur durch Verwendung der Präfixe verwendet werden. Tabelle 3-10 stellt die beiden Schreibweisen für die meist benötigten Verknüpfungen gegenüber, "x" und "y" stehen dabei für beliebige Terme oder Unteranfragen.

| Präfix-Angabe | boolesche Bedeutung | Bemerkung |
|---|---|---|
| +x +y | x AND y | |
| +(x y) | x OR y | Dokumente welche x und y enthalten werden höher bewertet |
| +x -y | x AND NOT y | |
| +x y | x AND PREFERABLE y | |

**Tabelle 3-10 Präfix-Schreibweise für boolesche Verknüpfungen**

## 3.8 Anfragebearbeitung - Implementierung

Der Parser für String-Anfragen ist im Package org.apache.lucene.queryParser definiert. Startpunkt ist hier die Klasse QueryParser, welche durch den Parser-Generator JavaCC ([27]) generiert wurde. Die zugehörigen JavaCC-Sourcen sind im selben Package abgelegt. Der Parser akzeptiert die in [5] beschrieben Syntax und setzt diese gemäß der ebenfalls dort definierten Semantik in eine entsprechende Objekt-Anfrage um. Von dieser Klasse ist für die Anwendung nur die statische Methode `parse` relevant. Diese erwartet drei Parameter `query`, `field` und `analyzer`. Der String-Parameter `query` gibt die String-Anfrage an. Der String-Parameter `field` definiert den Namen des Standard-Feldes, welches immer dann verwendet wird, wenn in der String-Anfrage kein Feld explizit angegeben wurde. Der Parameter `analyzer` gibt den zu verwendenden Analyse-Algorithmus an. Hier sollte derselbe Analyzer verwendet werden wie bei der Indizierung (siehe Abschnitt 3.6.1). Die in der String-Anfrage angegebenen Feld-Inhalte werden über diesen Analyzer analysiert und aufbereitet. Dieser Schritt ist wichtig, damit die Suchbegriffe in derselben Weise wie die Dokumente umgesetzt werden, so dass hier auch eine korrekte Suche über den Index erfolgen kann.

Die Klassen für eine Objekt-Anfrage sind im Package org.apache.lucene.search definiert. Im Rahmen dieser Arbeit werden nicht alle Klassen detailliert beschrieben. Ziel dieses Abschnitts ist die Verdeutlichung der Verwendung. Die genaue Beschreibung der Klassen ist in der JavaDoc ([2]) enthalten. Das Package definiert nicht nur die Klassen für eine Objekt-Anfrage sondern zusätzlich auch die Klassen für die Durchführung der Suche. Für die Objekt-Anfrage sind die von Query abgeleiteten Klassen relevant. Diese sind in Abbildung 3-9 dargestellt.

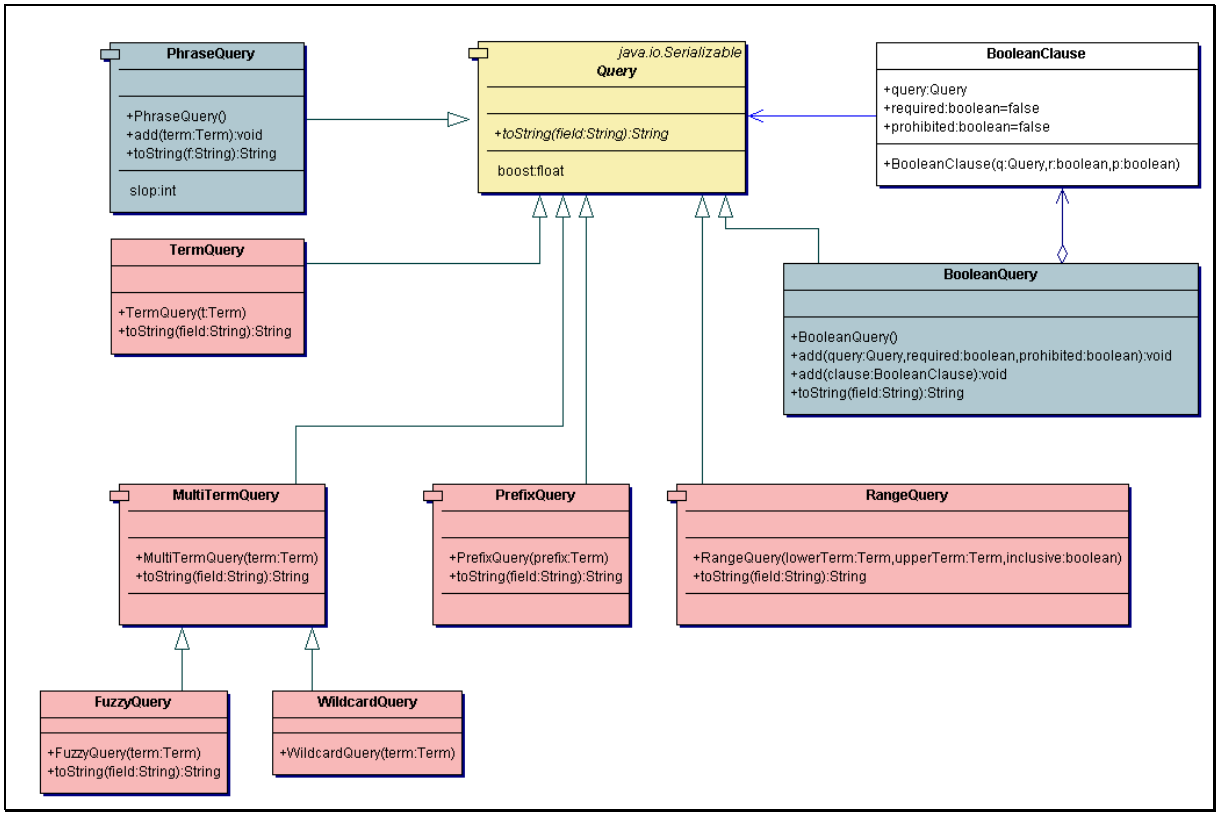


**Abbildung 3-9 Klassen für eine Objekt-Anfrage**

Die Abbildung zeigt nur die hier relevanten Attribute, Methoden und Verbindungen.

Die abstrakte Oberklasse Query (im Diagramm gelb hinterlegt) ist Basis für alle Klassen der Objekt-Anfrage. Sie enthält das Property "`boost`", welches den Faktor für die Berechnung der Relevanz angibt. Standardmäßig ist dieser auf 1.0 gesetzt. Jede Query besitzt eine Methode `toString(String field)`. Über diese Methode kann die Objekt-Anfrage in eine String-Anfrage umgewandelt werden. Dies ermöglicht eine Überprüfung während der Entwicklung. Der String-Parameter gibt das Default-Field an. Bei der Ausgabe werden nur die Feldnamen mitangegeben, welche nicht mit dem hier angegebenen Default-Field identisch sind. Würde diese String-Anfrage über den QueryParser mit demselben Default-Field analysiert, so würde dasselbe Objektmodell erzeugt werden.

Die Klasse BooleanClause baut eine Verbindung zwischen einer BooleanQuery und einer anderen Query auf. Dadurch können mehrere Teilanfragen durch eine BooleanQuery verbunden werden.

Alle anderen Query-Klassen arbeiten nur auf Termen. An dieser Stelle sei nochmals darauf hingewiesen, dass Objekt-Anfragen nicht durch den Analyse-Algorithmus aufbereitet werden. Es muss daher darauf geachtet werden, dass die Inhalte in einem zum Index passenden Format enthalten sind. Bei Bedarf kann dies durch Verwendung des Analyzer-Objektes aus Abschnitt 3.6.1 sichergestellt werden.

Die Klasse org.apache.lucene.index.Term ist aus Übersichtsgründen hier nicht dargestellt. Sie besteht im Wesentlichen aus zwei Strings welche den Feldnamen und den Inhalt angeben.

Es lassen sich zwei verschiedene Typen von Query-Klassen unterscheiden:

Bei den beiden blau hinterlegten Klassen PhraseQuery und BooleanQuery werden die Parameter durch Methoden angefügt. Es können hier beliebig viele Term- bzw. Query-Instanzen angefügt werden. Die restlichen (rosafarben hinterlegten) Query-Klassen erwarten die Parameter im Konstruktor. Die Anzahl von Termen ist hier festgelegt.

Die Namen der Klassen korrespondieren zu den in Abschnitt 3.7.1 genannten Anfragetypen.

Der Aufbau einer Objekt-Anfrage soll anhand eines Beispiels illustriert werden. Die Anfrage soll alle Dokumente liefern, welche folgende Bedingungen erfüllen:

- (1) Der Autor hat den Namen "Cutting" (der Initiator des Lucene-Projektes)
- (2) Der Text enthält das Wort "lucene"
- (3) Der Text enthält die Wörter "Query" und "search" zwischen denen maximal 5 andere Worte stehen dürfen (a), oder das Wort "Index" (b)

Der Autor sei im Index im Feld "author" abgelegt, der Text des Dokumentes im Feld "content". Als String-Anfrage (mit dem Feld "content" als Default-Feld) würde diese Anfrage dann `'author:cutting AND lucene AND ("Query search"~5 OR index)'` lauten.

Code-Snippet 3-2 baut diese Anfrage als Objekt-Anfrage auf.

```
1: TermQuery tQuery1 = new TermQuery(new Term("author", "cutting"));
2: TermQuery tQuery2 = new TermQuery(new Term("content", "lucene"));

3: PhraseQuery tQuery3a = new PhraseQuery();
4: tQuery3a.add(new Term("content", "query"));
5: tQuery3a.add(new Term("content", "search"));
6: tQuery3a.setSlop(5);

7: TermQuery tQuery3b = new TermQuery(new Term("content", "index"));

8: BooleanQuery tQuery3 = new BooleanQuery();
9: tQuery3.add(tQuery3a, false, false);
10:  tQuery3.add(tQuery3b, false, false);

11:  BooleanQuery tQuery = new BooleanQuery();
12:  tQuery.add(tQuery1, true, false);
13:  tQuery.add(tQuery2, true, false);
14:  tQuery.add(tQuery3, true, false);

15:  System.out.println(tQuery.toString("content"));
```

**Code-Snippet 3-2 Aufbau einer Objekt-Anfrage**

Die Zeilen 1 und 2 erzeugen die Teilanfragen (1) und (2). Die Teil-Anfrage (3) besteht wiederum aus zwei Teilanfragen. Die Anfrage (3a) wird in den Zeilen 3-6, die Teil-Anfrage (3b) in Zeile 7 erzeugt. Die Zeilen 8-10 kombinieren die beiden Teil-Anfragen (3a) und (3b). Der Parameter `required` wird hier mit false angegeben. Dies führt zunächst dazu, dass

keine der beiden Anfrage erfüllt sein muss. Die OR-Verknüpfung wird erst in Zeile 14 erzeugt, welche aussagt, dass die Teil-Anfrage 3 erfüllt sein muss. Die Zeilen 11-14 kombinieren nun die Teil-Anfrage (1), (2) und (3) zur gewünschten Objekt-Anfrage.

Zur Kontrolle wird in Zeile 15 die Objekt-Anfrage als String-Anfrage ausgegeben:

```
'+author:cutting +lucene +("query search"~5 index)'.
```

Diese Anfrage entspricht der "+/-"-Darstellung der gewünschten Anfrage.

## *3.9 Suche - Konzept*

Die Suche dient der Ermittlung eines Such-Ergebnisses, basierend auf der zuvor erzeugten Objekt-Anfrage. Dazu werden die Informationen im Index genutzt.

Die Suche kann auf einem einzelnen Index erfolgen. Es können aber auch auf mehrere Indizes verwendet werden. Diese werden dann sequentiell durchsucht und die Ergebnisse kombiniert.

Die Suche in einem Index erfolgt immer über alle vorhandenen Segmente. Sind mehrere Segmente vorhanden, so werden diese sequentiell durchsucht und die Ergebnisse anschließend kombiniert. An dieser Stelle wird der Zweck der in Abschnitt 3.5.3 angesprochenen Index-Optimierung nochmals deutlich.

### 3.9.1 Einschränken der Such-Ergebnisse

Oft wird die Anforderung gestellt, die Such-Ergebnisse unabhängig von der Anfrage weiter einzuschränken.

In Unternehmen sind beispielsweise viele Informationen vorhanden, jedoch darf nicht jeder diese Informationen lesen. Eine Such-Maschine in einem firmenweiten IntraNet muss diese Beschränkungen beachten. Eine Variante wäre sicherlich geschützte Informationen erst gar nicht zu indizieren. Dies verringert die Qualität der Suchmaschine jedoch gewaltig. Eine Filterung aufgrund der Zugriffsberechtigungen ist hier sicherlich die bessere Wahl. Auch eine Einschränkung basierend auf einem Zeitraum wird oft vorgenommen, insbesondere um neu eingefügte Dokumente zu durchsuchen.

Dokumente, welche aufgrund der Einschränkungen nicht im Such-Ergebnis erscheinen dürfen, werden im Folgenden ***beschränkte Dokumente*** genannt.

Für solche Einschränkungen sind in Verbindung mit Lucene prinzipiell drei Möglichkeiten vorhanden:

1. **Analyse bei der Ausgabe (Ausgabeanalyse):**
   Bei der Ausgabe wird für jedes gefundene Dokument geprüft ob dieses beschränkt ist. Ausgegeben werden nur die Dokumente, welche nicht beschränkt sind.

2. **Erweiterung der Anfrage (Anfrageerweiterung):**
   Die Such-Anfrage wird durch die Anwendung entsprechend erweitert, so dass diese nur nicht-beschränkte Dokumente findet.

3. **Anwendung eines Filters (Filterung):**
   Vor der Suche werden die beschränkten Dokumente identifiziert und der Suche ein

Filter übergeben. Bei der Durchführung der Suche wird diese Beschränkung beachtet, so dass im Such-Ergebnis keine beschränkten Dokumente vorhanden sind.

Diese Varianten sind in Abbildung 3-10 dargestellt.

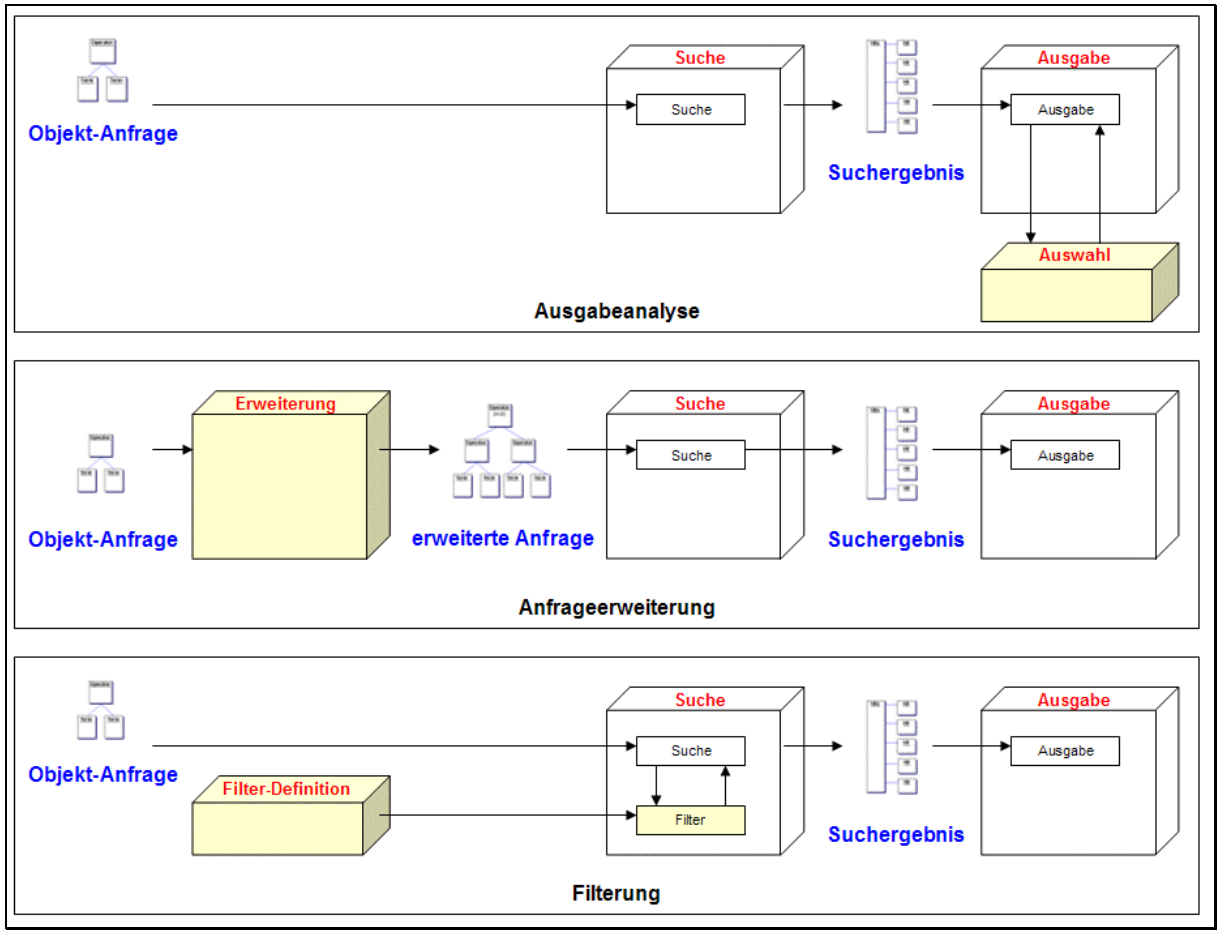


**Abbildung 3-10 Varianten der Beschränkung von Suchergebnissen**

Die Variante *Ausgabeanalyse* kann bei jeder Suchmaschine und unabhängig von den im Index verfügbaren Daten angewendet werden. Jedoch erfordert sie eine komplexe Aufbereitung der Such-Ergebnisse um die Anzahl der Treffer und den gezielten Zugriff auf Teilmengen bestimmter Größen, wie er bei der Ausgabe mehrerer Ergebnis-Seiten erforderlich ist, zu ermöglichen.

Die Variante *Anfrageerweiterung* ist nur dann möglich, wenn die im Index verfügbaren Daten für eine Entscheidung ausreichen. Dafür kann das Such-Ergebnis direkt für eine Ausgabe verwendet werden, da es nur nicht-beschränkte Dokumente enthält.

Bei der Variante *Filterung* können wie bei der Ausgabeanalyse auch Informationen verarbeitet werden, welche nicht im Index vorhanden sind. Gleichzeitig arbeitet sie wie die Anfrageerweiterung vor der Erstellung der Such-Ergebnisse. Die Such-Ausgabe enthält also auch bei dieser Variante nur nicht-beschränkte Dokumente und kann daher direkt ausgegeben werden.

Da die Ausgabeanalyse aufgrund der Datenmenge meist sehr aufwändig ist, ist diese den anderen Varianten klar unterlegen und wird an dieser Stelle nicht weiter vertieft.

Die Anfrageerweiterung und die Filterung werden im Folgenden näher beschrieben.

Ein Filter wird über eine geordnete Bit-Folge definiert, wobei jedes Bit ein Dokument repräsentiert. Der Wert des jeweiligen Bits definiert, ob dieses Dokument im Such-Ergebnis vorkommen darf oder nicht. Das Such-Ergebnis wird also auf diejenigen Dokumente eingeschränkt, welche den Such-Bedingungen entsprechen und gleichzeitig aufgrund des Filters zur Ausgabe berechtigt sind. Die Länge dieser Bit-Folge ist offensichtlich durch die Anzahl der im Index vorhandenen Dokumente gegeben. Sortiert ist diese Folge nach den internen Dokument-Nummern.

Lucene liefert einen Filter mit, welcher einen Wertebereich bei Datumsangaben in einem Feld auswertet. Dadurch kann eine Einschränkung z.B. auf Basis eines Veröffentlichungsdatums erfolgen. Dies kann verwendet werden um Seiten wie "Veröffentlichungen der letzten 10 Tage" zu erstellen.

Eine Einschränkung des Zeitraumes (Feld *modified*) ist, vorausgesetzt ein entsprechendes Datum ist im Index abgelegt, auch einfach durch eine Bereichssuche möglich:

```
"(<Original-Anfrage>) AND (modified:[<Beginn>-<Ende>])"
```

Auch wenn die beiden Alternativen keinen Unterschied bezüglich der Funktionalität haben, ist durchaus ein Unterschied in der Performanz zu beobachten. Die Bereichssuche benötigt mehr Zeit welche stärker steigend mit der Größe der Zeitspanne ist. Der Grund dafür liegt in der Implementierung: Für die Bereichssuche werden alle einzelnen Terme im Index, welche im angegebenen Bereich liegen mit der Anfrage verknüpft. Eine Bereichsanfrage "N1-N5" wird umgewandelt in die Anfrage

```
"xyz AND (feld:N1 OR feld:N2 OR ... OR feld:N5)".
```

Dies führt zu sehr komplexen Abfragen und kann bei genügender Größe auch Speicher-Probleme verursachen. Die Verwendung des Filters verursacht diese Probleme nicht.

### 3.9.2 Berechnung der Relevanz

Jedes Dokument im Such-Ergebnis wird mit einer Zahl zwischen 0 und 1 (*Relevanz*, siehe Abschnitt 2.5) versehen, welche basierend auf der Suchanfrage und den Fundstellen in dem jeweiligen Dokument errechnet wird. Das Such-Ergebnis wird nach dieser Relevanz sortiert.

Die Berechnung der Relevanz berücksichtigt u. a. folgende Faktoren:

- Eine Fundstelle in einem 5 Wörter langen Text führt zu einer höheren Relevanz in einem 500 Wörter langen Text
- Mehr Fundstellen in einem Text führen zu einer höheren Relevanz
- Zwei gesuchte Terme in einem Dokument führen zu einer höheren Relevanz als nur ein gesuchter Term

- Ein höher bewerteter Term (vgl. Abschnitt 3.7.1) führt zu einer höheren Relevanz als ein niedriger bewerteter Term.

- In der Lucene Version 1.3 führt ein gesuchter Term in einem höher bewerteten Dokument (vgl. Abschnitt 3.5.4) zu einer höheren Relevanz als derselbe Term in einem ansonsten äquivalenten Dokument

Der Algorithmus zur Berechnung der Relevanz wird in Abschnitt 3.10.2 beschrieben.

## 3.10 Suche - Implementierung

Die Suche ist im Package org.apache.lucene.search implementiert. Abbildung 3-11 zeigt die Zugriffsklassen, wobei nicht relevante Methoden ausgeblendet wurden.

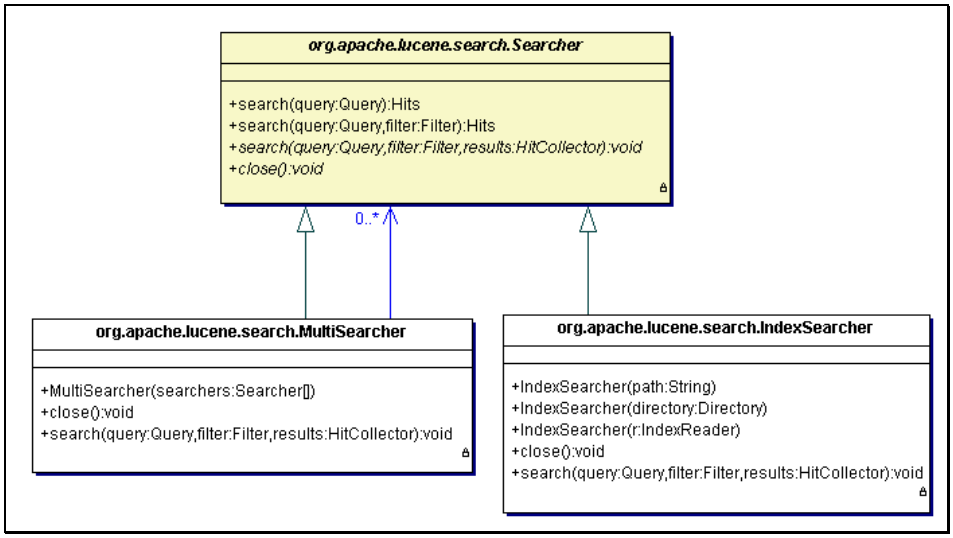


**Abbildung 3-11 Zugriffsklassen für die Suche**

Die abstrakte Superklasse Searcher bietet die beiden für die meisten Suchanfragen geeigneten Methoden `search`. Diese erwarten entweder eine Objekt-Anfrage oder eine Objekt-Anfrage in Verbindung mit einem Filter (siehe Abschnitt 3.10.1). Diese geben das Such-Ergebnis als ein org.apache.lucene.search.Hits-Objekt zurück. Dieses wird in Abschnitt 3.12 beschrieben.

Für diese Klasse existieren zwei Implementierungen. Die Klasse IndexSearcher ist für Suchen in einem einzelnen Index vorgesehen. Sollen mehrere Indizes durchsucht werden, so muss für jeden dieser Indizes ein IndexSearcher erzeugt werden. Aus diesen IndexSearcher-Objekten wird dann ein MultiSearcher erzeugt.

Der IndexSearcher hat drei Konstruktoren, welche sich in der Angabe des Index unterschieden.

Die abstrakte Methode `search(Query, Filter, HitCollector)` der Klasse Searcher dient hauptsächlich der Implementierung der konkreten Suche. Der HitCollector wird bei jedem Fund eines Dokumentes für das Such-Ergebnis benachrichtigt. Für die Anwendung ist dies aber nicht relevant und soll an dieser Stelle auch nicht ausgeführt werden.

Nach Abschluss der Suche werden durch die Methode `close` verwendete Ressourcen wieder freigegeben.

### 3.10.1 Einschränkung der Such-Ergebnisse

Die Einschränkung der Such-Ergebnisse durch Filterung erfolgt durch Implementierungen der abstrakten Klasse org.apache.lucene.search.Filter. Eine Instanz dieser Implementierung wird bei der Suche mit angegeben. Abbildung 3-12 zeigt diese Klasse in UML-Darstellung.

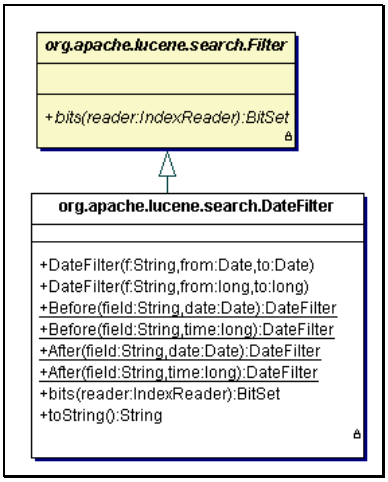


**Abbildung 3-12 Klassen für eine Filterung der Such-Ergebnisse**

Die Klasse erfordert nur die Implementierung der abstrakten Methode `bits`. Die Methode muss eine java.util.BitSet-Instanz zurückgeben. Dieses BitSet ist logisch ein Array aus Bits. Das Array ist nach aufsteigenden internen Dokument-Nummern sortiert und gibt jeweils mit einem "true" an, dass das zugehörige Dokument im Such-Ergebnis erscheinen darf.

Der übergebene IndexReader dient dem Zugriff auf den Index. Aus dem IndexReader kann die Anzahl der Dokumente ermittelt werden, sowie die internen Dokument-Nummern.

Die einzige mitgelieferte Implementierung ist die Klasse DateFilter. Diese nimmt eine Filterung basierend auf einem Feld mit einer Datumsangabe vor. Zu beachten ist, dass der Inhalt dieses Feldes bei der Indizierung wie in Abschnitt 3.4 beschrieben in das interne Datums-Format umgewandelt werden muss. Die Klasse bietet zwei Konstruktoren für die Angabe des Zeitraumes an. Zusätzlich stellt sie die statischen Methoden `Before` und `After` bereit. Sowohl bei den Konstruktoren als auch bei den statischen Methoden muss der Feldname mit angegeben werden.

Zu beachten ist, dass für eine Suche nur ein einzelnes Filter-Objekt übergeben werden kann. Sollen mehrere Filter angewendet werden, so muss dies extern implementiert werden, beispielsweise durch ein Mischen der BitSets.

### 3.10.2 Berechnung der Relevanz

Die Relevanz wird durch Implementierungen der Klasse org.apache.lucene.search.Scorer berechnet. Die genaue Implementierung dieser Klassen soll an dieser Stelle nicht dargestellt werden. Bei Bedarf sei hier auf die Lucene-Sourcen verwiesen, welche auf [1] bereitgestellt

werden. Stattdessen konzentriert sich dieser Abschnitt auf den durch die Klassen implementierten Algorithmus.

Nach [3] wird die Relevanz nach folgender Formel berechnet:

$$score_d = \sum_{alle\_Terme\_t} \left( tf_q * \frac{idf_t}{norm_q} * tf_d * \frac{idf_t}{norm_{d\_t}} * boost_t \right) * coord_{q\_d}$$

wobei

| | |
|---|---|
| $score_d$ | Relevanz für Dokument d |
| $tf_q$ | $\sqrt{\#Vorkommnisse\_von\_t\_in\_der\_Anfrage}$ |
| $idf_t$ | $\log\left(\frac{numDocs}{docFreq_t}+1\right)$ |
| $numDocs$ | Anzahl der Dokumente im Index |
| $docFreq_t$ | Anzahl der Dokumente, welche t enthalten |
| $norm_q$ | $\sqrt{\sum_{alle\_Terme\_t}\left((tf_q * idf_t)^2\right)}$ |
| $tf_d$ | $\sqrt{\#Vorkommnisse\_von\_t\_in\_Dokument\_d}$ |
| $norm_{d\_t}$ | $\sqrt{\#Tokens\_in\_d\_im\_selben\_Feld\_wie\_t}$ |
| $boost_t$ | In der Anfrage angegebener Faktor für Term t (Standard: 1.0) |
| $coord_{q\_d}$ | $\frac{\#Terme\_in\_Anfrage\_und\_Dokument}{\#Terme\_in\_Anfrage}$ |

Dieser Algorithmus hat Schwächen bei sehr kleinen Index-Größen. Da die Wahrscheinlichkeit sehr groß ist, dass der Term in allen Dokumenten vorkommt, wird der Faktor $idf_t$ kleiner als 1. Dadurch wird Faktor $\frac{idf_t}{norm_{d\_t}} << 1$ und beeinflusst extrem die Relevanz. Dies führt dazu, dass auch 100%-ig exakte Ergebnisse, wie bei einer Suche nach dem exakten Dokument-Inhalt, eine sehr geringe Relevanz besitzen. Desweiteren führen semantisch äquivalente Anfragen, wie "tree" und "tree OR tree" zu sehr unterschiedlichen Relevanz-Werten. Eine Stabilisierung erfolgt, wenn $idf_t$ den Wert 1 erreicht oder überschreitet. Dies ist gegeben bei $\frac{numDocs}{docFreq_t} \geq 9$, oder anders ausgedrückt, wenn die Suchbegriffe in höchstens 10% aller Dokumente enthalten sind.

## 3.11 Ergebnispräsentation - Konzept

Die *Ergebnispräsentation* hat zur Aufgabe, das bei der *Suche* erzeugte *Such-Ergebnis* dem Anwender zu präsentieren.

Das Such-Ergebnis kann als eine Liste mit wahlfreiem Zugriff angesehen werden. Die Liste enthält alle gefundenen Dokumente zusammen mit deren Relevanz-Werten. Durch den wahlfreien Zugriff ist eine Aufteilung der Darstellung der Such-Ergebnisse in mehrere Seiten möglich, ohne dass bei einem Seiten-Wechsel eine erneute Suche durchgeführt werden muss. Die Dokumente enthalten die Namen und Inhalte aller Felder, welche im Schritt *Aufbereitung des Suchraums* mit der Option *store* angelegt wurden.

## 3.12 Ergebnispräsentation - Implementierung

Kern der Daten für die Ergebnispräsentation ist die Klasse org.apache.lucene.search.Hits. Diese ist zusammen mit den anderen für die Präsentation relevanten Klassen in Abbildung 3-13 dargestellt.

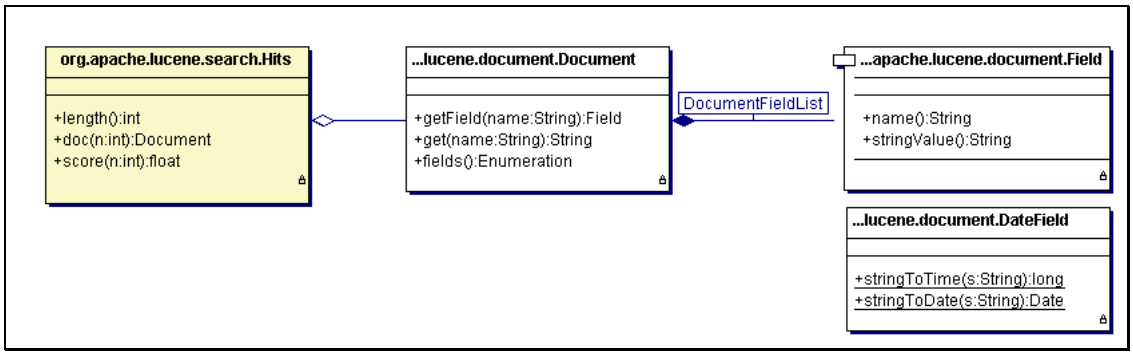


**Abbildung 3-13 Relevante Klassen für die Ergebnispräsentation**

Im Diagramm sind nur die Methoden dargestellt, welche für die Ergebnispräsentation wichtig sind.

Über die Methode Hits.`length` kann die Anzahl der gefundenen Dokumente ermittelt werden. Die Methode Hits.`doc` dient dem Zugriff auf ein bestimmtes Dokument, die Methode Hits.`score` gibt die zugehörige Relevanz zurück. Als Parameter wird hier die Nummer des Dokumentes angegeben. Diese liegt zwischen 0 und Hits.`length`() - 1.

Die Klassen Document, Field und DateField wurden bereits in Abschnitt 3.4 vorgestellt.

Für die Ergebnispräsentation kann entweder über die Methode Document.`getField` auf das zugehörige Field-Objekt zugegriffen werden, oder über die Methode Document.`get` direkt auf den Inhalt des entsprechenden Feldes. Zu beachten ist hier, dass bei der Angabe der Feld-Namen die Groß-Kleinschreibung beachtet werden muss.

Felder welche Datumsangaben enthalten, wurden bei der Anlage in Abschnitt 3.4 in ein internes Format konvertiert. Die Klasse DateField bietet mit den Methoden `stringToTime` und `stringToDate` nun die Möglichkeit, diese wieder in ein Format zu konvertieren, welches mit Standard-Java-Mitteln dem Benutzer lesbar angezeigt werden kann.

## 3.13 Bewertung des Frameworks

Lucene vereint mächtige Funktionen mit einer einfachen Nutzung. Gleichzeitig ist Lucene für ein breites Spektrum an Einsatzgebieten geeignet. Einzelne Vorkonfigurationen (z.B. die in Abschnitt 3.5.1 angesprochene Längenbegrenzung von Feldern) können jedoch bei einer naiven Implementierung die Qualität einer Suche erheblich einschränken. Die Zweckmäßigkeit der Methode um hohen Hauptspeicherbedarf zu begrenzen, in dem die Länge eines Feldes beschnitten wird, ist zweifelhaft, da sie die Funktionalität der Suche einschränkt. Dieser Punkt ist aus Anforderungssicht der größte Nachteil von Lucene.

Die Architektur von Lucene wird in [22] als "bemerkenswertes Beispiel für flexibles Softwaredesign" bezeichnet. Sie hat jedoch auch ihre Schwachstellen, von denen an dieser Stelle nur die Wichtigsten genannt werden:

Die Anpassungsfähigkeit ist durch die fehlende Verwendung von Interfaces begrenzt. In der bestehenden Architektur können beispielsweise die Klassen für eine Objekt-Anfrage nicht durch eigene Implementierungen ersetzt werden. Die ausschließliche Verwendung abstrakter Klassen verhindert auch die komplette Austauschbarkeit der einzelnen Komponenten (wie der Indizierungskomponente). Zusätzlich sind viele Klassen als final deklariert, was auch eine Erweiterung durch Vererbung verhindert. Insbesondere bei der Kopplung der einzelnen Komponenten würde die Umsetzung des Design-by-Interface-Paradigmas ([39]) einige Einschränkungen verhindern.

Löschungen erfolgen entgegen der durch die Klassen-Namen suggerierten Aufteilung nicht durch den IndexWriter, sondern durch den IndexReader. Dies hätte zumindest durch den Einsatz package-lokaler Methoden als Implementierungsdetail verborgen werden können.

Lucene setzt in großem Maße öffentliche Attribute ein. Attribute sollten allerdings immer klasseninterne Sichtbarkeit besitzen und den Zugriff über entsprechende Accessor-Methoden kapseln. Durch die direkte Zugreifbarkeit können beispielsweise keinerlei Prüfungen auf die Einhaltung von Randbedingungen erfolgen.

Zusätzlich verstoßen sehr viele Methoden gegen die international anerkannten Java Code-Conventions von Sun ([42]). Insbesondere bei einem OpenSource-Projekt muss ein hoher Wert auf die Einhaltung internationaler Konventionen und der Beachtung allgemein üblicher Verfahrensweisen gelegt werden, da viele Personen außer den Entwicklern den Code lesen und auch bearbeiten. Die Namen statischer Methoden beginnen beispielsweise bei Lucene mit einem Großbuchstaben. Gemäß den Konventionen muss eine Methode mit einem Kleinbuchstaben beginnen. Die Verletzung führt hier leicht zu einer Verwechslung mit einem Klassennamen und damit mit einem Konstruktor. Die Konventionen sagen auch aus, dass ein Methodenname mit einem Verb beginnt, welcher die Funktion beschreibt. Gefolgt wird er durch eine entsprechende Angabe des Gegenstands der Funktion. In sehr vielen Fällen fehlt entweder das Verb oder der Gegenstand der Methode. Die Folge davon sind nicht-sprechende Methodennamen. Diese bergen eine große Gefahr der Missdeutung und eigentlich unnötige Konsultationen der Dokumentation. Als Beispiel sei hier die Methode Hits.`get(int)` genannt. Dass diese ein Dokument zurückgibt wäre durch die üblichere Bezeichnung Hits.`getDocumentAt(int)` leicht zu erkennen. Die statischen Methoden der Field-Klasse sind ein weiteres Beispiel. Eine Bennung von Field.`createKeyword(pName, pValue)` würde auch ohne weitere Dokumentation die Wirkungsweise verdeutlichen.

Die Benennung von Variablen und Parametern führt zu Verständnisproblemen. Lucene setzt viele nicht-sprechende "1-Zeichen"-Namen ein, wodurch der Code umständlich zu lesen wird. Als Beispiel sei hier die Klasse org.apache.lucene.IndexWriter genannt, welche im Abschnitt 0 in Abbildung 3-8 dargestellt ist. Im Konstruktor wird als Name für den Parameter des Typs Analyzer hier lediglich ein "`a`" verwendet. Eine bessere Bezeichnung wäre hier "`pAnalyzer`", wobei das Präfix "`p`" auf einen Parameter hindeutet. Insbesondere bei den für den Anwender wichtigen Klassen stellt dies ein Problem dar, da diese von sehr vielen Entwicklern direkt verwendet werden.

Eine große Gefahrenquelle ist die Überdeckung von Variablen. So definiert beispielsweise die Klasse org.apache.lucene.analysis.standard.StandardAnalyzer in der Methode `next` eine Variable mit dem Namen `token`. Die Klasse enthält allerdings ein gleichnamiges Attribut, welches auch den gleichen Typ hat. So ist die Verwechslungsgefahr groß, durch denselben Typ kann diese auch nicht durch den Compiler entdeckt werden.

Trotz der hier dargestellten Kritik, welche sich hauptsächlich auf die innere Architektur bezieht, ist Lucene eine mächtige OpenSource-Software die sich durchaus mit kommerziellen Produkten messen kann.

Kapitel 4

# FIA – Föderale Informations Architektur

Dieses Kapitel beschreibt die Föderale Informations-Architektur (FIA). Ziel dieser Arbeit ist die Konzeption einer Erweiterung dieser Architektur um eine Volltext-Suchfunktion. Dem Leser wird an dieser Stelle ein Überblick über die bestehende Anwendung und deren für die Arbeit relevanten Komponenten gegeben. Die Ansatzpunkte für die geforderte Suche werden detailliert vorgestellt. Weitere Eigenschaften werden im Kontext der Umsetzung der Volltextsuche in Kapitel 5 beschrieben.

## 4.1 Motivation

Ein Überblick über die Motivation und Umsetzung ist in [13] beschrieben. Dieser Abschnitt fasst die wichtigsten Informationen zusammen.

Der Maschinen- und Anlagebau setzt oft über Jahre gewachsene Entwicklungsprozesse ein. Eine Wiederverwendung bestehender Komponenten in neuen Projekten erfolgt durch Kopieren von Daten. Dies hat eminente Nachteile. Insbesondere werden vorhandene Fehler kopiert und es erfolgt weder eine Standardisierung noch eine projektübergreifende Abstimmung. Ergebnis sind Mehrfachentwicklungen und mangelnde Qualität. Das Ziel ist nun diese Form der Wiederverwendung zu vermeiden und durch eine baukastenbasierte Wiederverwendung zu ersetzen. Dies wird in Abbildung 4-1 illustriert.

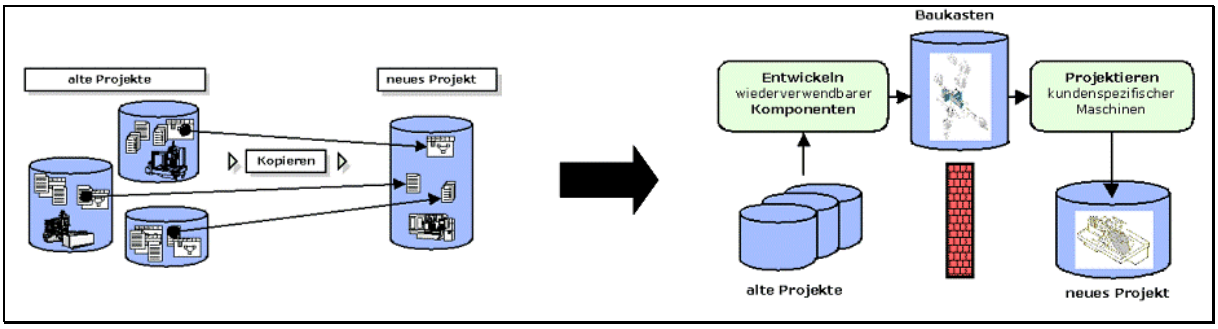


**Abbildung 4-1 Methodisches Ziel der FIA ([13 m. Ä.])**

Die links gezeigte Wiederverwendung durch Kopieren soll in die rechts gezeigte baukastenbasierte Wiederverwendung überführt werden. Dies erfolgt dadurch, dass aus bestehenden Projekten wiederverwendbare Komponenten extrahiert und in einen Baukasten eingefügt werden. Kundenspezifische Projekte werden aus den vorhandenen Komponenten durch Projektierung (Kombination und Konfiguration) zusammengestellt. Fehlerbereinigungen und Neuentwicklungen erfolgen zentral im Baukasten und können daher von allen Projekten verwendet werden. Dieses Vorgehen hat wesentliche Vorteile, insbesondere werden durch den Aufbau projektneutraler Firmenstandards und zentraler Repräsentation des Firmen-KnowHows eine Reduzierung der Durchlaufzeiten und damit der

Fertigungskosten erreicht. Die Wiederverwendung erfolgt jedoch nur in Teilbereichen wie der Software-Entwicklung. Der Grund dafür liegt in der Natur der Komponenten. Diese sind komplexe mechatronische Einheiten, welche nicht als abgeschlossene physische Einheit existieren. Sie entstehen durch die Zusammenarbeit verschiedener Ingenieurs-Disziplinen, welche mit unterschiedlichen Softwaresystemen arbeiten, deren Datenablagen nicht auf eine derartige Wiederverwendung ausgelegt sind.

Auch wenn Baukastensysteme in den Köpfen der Entwickler existieren, sind aufwändige manuelle Routinetätigkeiten (z.B. Kopieren) erforderlich, um kundenspezifische Maschinen zu entwerfen. In der Vergangenheit wurden firmenspezifische Integrationslösungen entwickelt, um diese Routinetätigkeiten zu reduzieren. Dabei wird allerdings keine abteilungs- und anwendungsübergreifende Integration der Datenbestände erreicht.

Die Idee welche hinter der FIA steckt ist nun, die bisherige Vorgehensweise durch eine modellbasierte Entwicklung (model-driven-Engineering) zu ersetzen. Abbildung 4-2 illustriert diese.

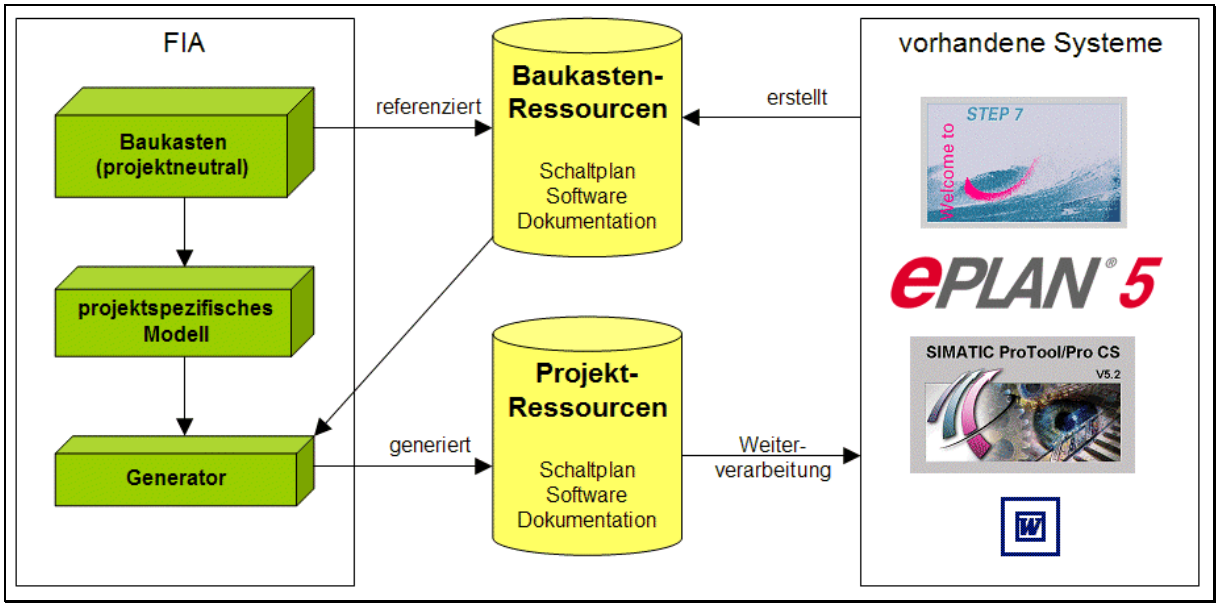


**Abbildung 4-2 modellbasierte Entwicklung mit der FIA**

Mit den vorhandenen Systemen werden die parametrisierbaren Daten der Komponenten (Software, Dokumentation, Schaltpläne) des Baukasten erstellt. Die FIA modelliert auf abstrakter Ebene diese mechatronischen Komponenten (Baukasten) und definiert die verfügbaren Parameter. Durch einen Modellierungsschritt werden neue Maschinenstrukturen unter Verwendung von Komponenten aus dem Baukasten erstellt und die verfügbaren Parameter mit Werten belegt. Ein Generator erzeugt aus den referenzierten Dateien und den Modellinformationen die projektspezifischen Ressourcen, welche durch die bestehenden Anwendungen weiterverarbeitet (z.B. gedruckt) werden.

Eine wichtige Voraussetzung ist, dass die Entwicklung auch ausschließlich durch Änderungen an den Quellen oder dem Modell erfolgen, so dass redundante Arbeiten systematisch vermieden werden und die Konsistenz mit den Daten in der FIA gewährleistet wird. Die Tätigkeiten verlagern sich also auf die Erstellung eines durchgängigen und ausdrucksstarken formalen Modells.

## 4.2 Umsetzung

Für die FIA wurde eine Modellierungsmethodik und ein dazu passendes Werkzeug entwickelt, welches die Erstellung der Modelle ermöglicht. Das Werkzeug basiert auf dem OpenSource-Framework Eclipse, welches eine komponentenbasierte universelle Plattform für die Erstellung von Entwicklungsplattformen bereitstellt. Die Plattform wird um Komponenten für diese modellbasierte Entwicklung erweitert. Abbildung 4-3 zeigt den Grundaufbau der FIA.

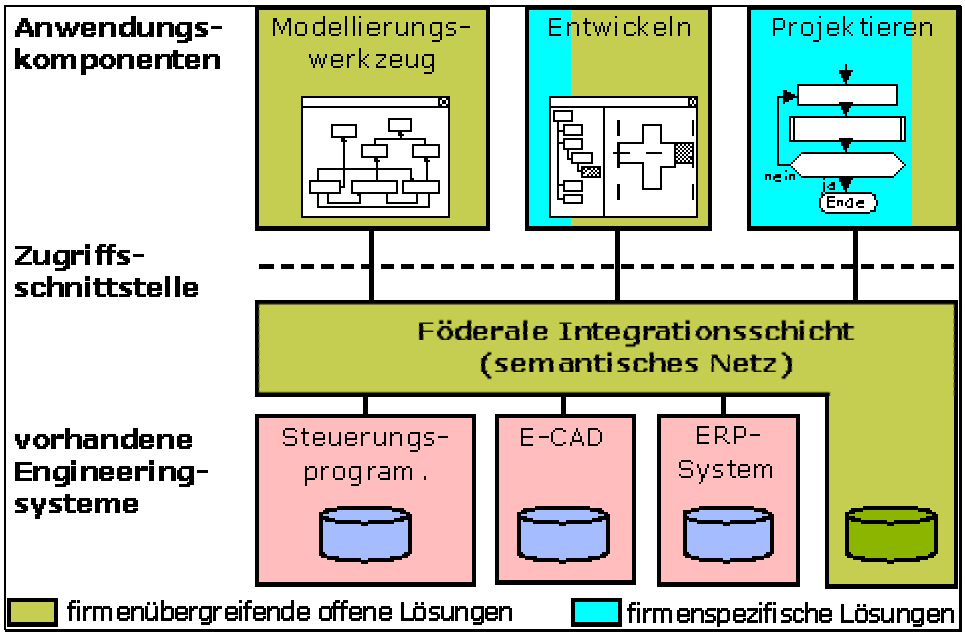


**Abbildung 4-3 Aufbau der FIA [13]**

Die meisten Daten der einzelnen Komponenten werden in den vorhandenen Engineering-Systemen (im unteren Bereich der Abbildung) bearbeitet und abgelegt. Über diese vorhandenen Systeme wird ein semantisches Netz gelegt, welches auf die in diesen Systemen abgelegten Daten referenziert. Diese Form der Integration verschiedener spezialisierter Anwendungen wird als *föderale Integration* bezeichnet. Ein firmenneutrales Modellierungswerkzeug ermöglicht den Aufbau von Verbindungen zwischen den verschiedenen Komponenten. Die FIA benutzt eine eigene Datenbank zur Speicherung der Modellstruktur, da die vorhandenen Systeme nicht auf diese Arten der Verbindungen ausgelegt sind. Die Entwicklung neuer Komponenten kann durch eine weitgehend firmenunabhängige Anwendungskomponente erfolgen. Neue Maschinen werden durch ein als *Projektieren* bezeichnetes, hochgradig firmenspezifisches, Verfahren modelliert.

Die Anwendung stellt die benötigten Komponenten für die Modellierung, Entwicklung und Projektierung zur Verfügung. Die Entwicklung und insbesondere die Projektierung ist firmenabhängig. Daher werden diese Komponenten firmenspezifisch entwickelt und bilden dadurch die speziellen Anforderungen ab.

## 4.3 Technischer Aufbau

Die FIA ist in 5 Schichten unterteilt, wie Abbildung 4-4 in einer Übersicht zeigt.

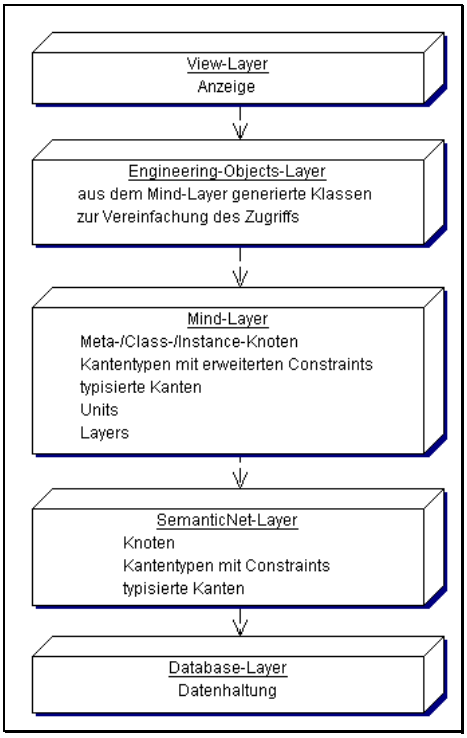


**Abbildung 4-4 Schichten in der FIA**

Der *Database-Layer* dient der Verwaltung der Daten durch Anbindung einer Datenbank. Diese Anbindung ist austauschbar, um den Einsatz verschiedener Standardprodukte (sowohl relational wie IBM DB/2 als auch objektorientiert wie Poet) zu ermöglichen.

Der *SemanticNet-Layer* ist primär für die transaktionssichere Persistenz in der darunterliegenden Datenbank verantwortlich. Er unterscheidet Knoten, typisierte Kanten und Kantentypen. Jeder Knoten enthält ein einzelnes String-Attribut. Die Kantentypen können mit einfachen Randbedingungen (Constraints) belegt werden, welche beispielsweise die Kardinalität einer solchen Kante definieren. Der Zugriff erfolgt wahlweise durch eine Navigation oder deklarative Anfragen über SNQL ([41]), einer spezialisierten XQuery-ähnlichen, mengenbildenden Abfragesprache, erfolgen.

Der *Mind-Layer* führt fachliche Abstraktionsebenen durch eine Typisierung der Knoten in Meta-, Class- und Instance-Knoten (siehe Abschnitt 4.4) ein. Diese Abstraktionsebenen werden analog als Meta-, Class- und Instance-Layer bezeichnet. Die Kantentypen werden um Constraints erweitert, welche die Definition erlaubter Abhängigkeiten zwischen diesen Ebenen erlauben. Units bauen zusätzlich über den Knoten und Kantentypen eine baumartige Hierarchie auf, welche primär organisatorisch motiviert ist.

Der *Engineering-Objects-Layer* dient dem einfachen Zugriff auf den Mind-Layer. Er definiert hierzu aus dem Mind-Layer generierte Klassen.

Der *View-Layer* ist schließlich für die Anzeige der Netze verantwortlich.

## 4.4 Fachlicher Aufbau

Die Anwendung unterscheidet im Mind-Layer die drei Abstraktionsebenen "Meta-", "Class-" und "Instance-Layer", welche durch Abbildung 4-5 illustriert werden.

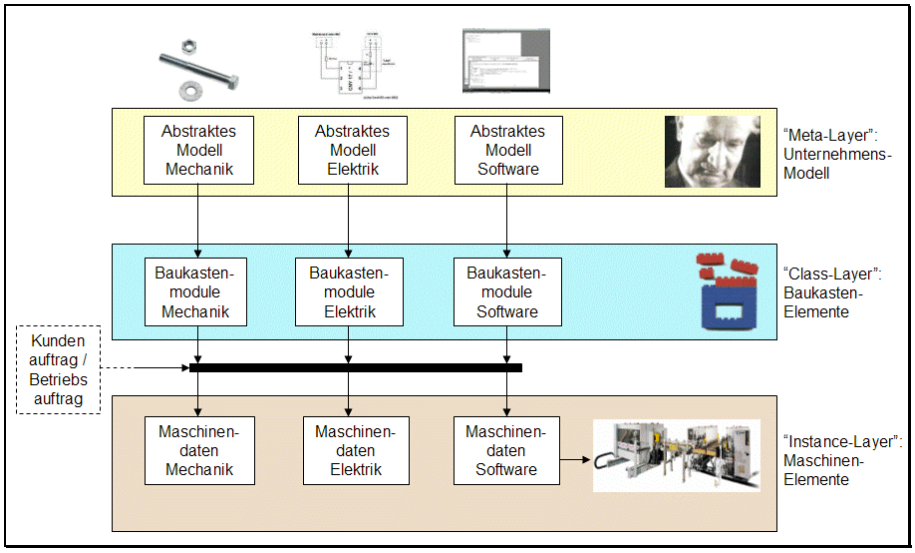


**Abbildung 4-5 Fachliche Ebenen im Mind-Layer**

Der ***Meta-Layer*** definiert abstrakte Modelle für die drei Disziplinen *Mechanik*, *Elektrik* und *Software*. Die Struktur dieser Modelle legen für jeden einzelnen Knoten fest, ob dieser einen Verweis auf eine externe Datei definiert, zusammen mit den nötigen Informationen zur Bestimmung des Ablageortes. Der ***Class-Layer*** konkretisiert diese Modelle durch Instanziierung zu einzelnen Baukastenmodulen. Aufgrund eines Kundenauftrags werden im ***Instance-Layer*** schließlich konkrete Maschinen aus diesen Komponenten modelliert.

## 4.5 Vererbung von Knoteninhalten

Das semantische Netz unterstützt die Vererbung von Knoteninhalten. Das bedeutet, dass ein Knoten ohne Inhalt semantisch den Inhalt des zugehörigen Knotens der nächsthöheren Abstraktionsebene (Class- bzw. Meta-Knoten) enthält (erbt). Dieser Inhalt wird als ***semantischer Inhalt*** bezeichnet. Abbildung 4-6 zeigt die Darstellung in der Anwendung und definiert die für die Beispiele verwendete Notation.

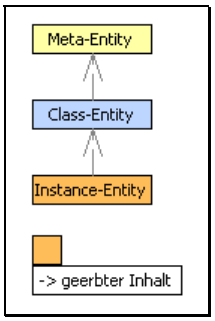


**Abbildung 4-6 Notation für diesen Abschnitt**

Die farbig hinterlegten Kästen visualisieren die Knoten des Netzes. Die verwendete Hintergrundfarbe zeigt die zugehörige Abstraktionsebene (*Meta*, *Class* oder *Instance*). Der Text in den Kästen definiert den Knoteninhalt. Zwischen den Abstraktionsebenen werden Instanziierungsbeziehungen durch grau gestrichelte Linien dargestellt. Bei leeren Knoten wird in den Beispielen der semantische Inhalt durch ein angehängtes Kommentarfeld angegeben.

Abbildung 4-7 verdeutlicht die Motivation der Knoteninhalts-Vererbung an einem Ausschnitt aus einem realen Modell.

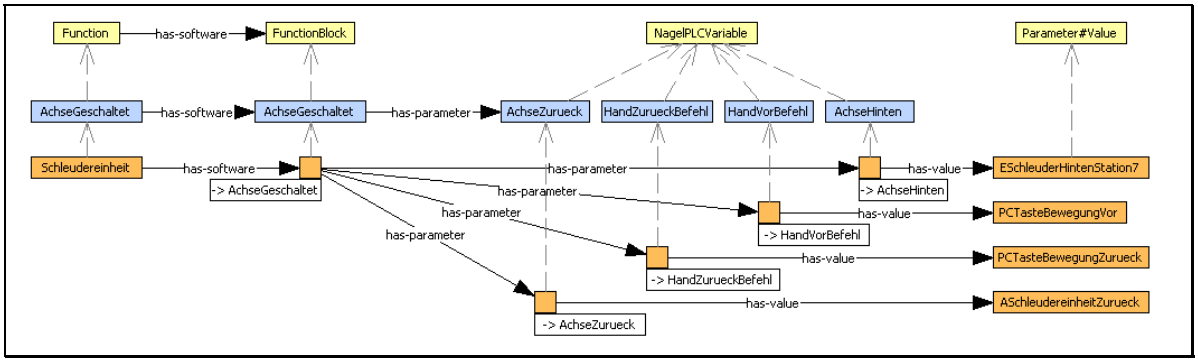


**Abbildung 4-7 Knoten-Inhalts-Vererbung in der FIA**

Die leeren Instance-Knoten (in der Abbildung orange) übernehmen semantisch den Inhalt der zugehörigen Class-Knoten (in der Abbildung blau). Dieses Vorgehen hat mehrere Vorteile. Beispielsweise kann der Name eines Parameters geändert werden ohne alle Verwendungen zu aktualisieren. Des Weiteren ist bei der Anlage einer neuen Maschine, was durch Erstellung des entsprechenden Instanzen-Modells erfolgt, kein fehlerträchtiges Abschreiben der entsprechenden Parameternamen erforderlich. Weitere Ausführungen zur Vererbung der Knoteninhalte sind bei der Umsetzung in Abschnitt 5.3.5 gegeben.

## 4.6 Motivation für eine Suchfunktion

Die bestehende Anwendung erlaubt die Navigation im Netz und die Auswahl von Teilnetzen durch zwei Verfahren:

- Ausgehend von einem Einstiegsknoten ist eine prozedurale Navigation im Netz und damit die Auswahl vom Teilnetzen möglich. Dieser Zugriff setzt die Kenntnisse über die verwendeten Relationstypen, deren Semantik sowie die Struktur des Netzes voraus.

- Basierend auf der Definition von Relationstypen wird eine deklarative Anfrage über SNQL an das Netz gestellt, die zu einer Ergebnismenge von Knoten führt. Um diese Anfrage zu definieren sind Grundkenntnisse über die Struktur des Netzes erforderlich

Diese Möglichkeiten reichen für eine vollständige Durchsuchbarkeit nicht aus. Dies ist insbesondere durch folgende Faktoren begründet:

- Die Verfahren berücksichtigen nur die Informationen welche direkt im Netz abgelegt sind. Die Inhalte referenzierter Dateien können nicht herangezogen werden.

- Die Struktur des Netzes muss - zumindest grob - bekannt sein.
- Die Verfahren bieten zu wenig Funktionalität für die Definition universeller Suchanfragen.

Ist der Anwender auf der Suche nach einem bestimmten Inhalt eines Knotens (z.B. Teil eines Projektnamens) ohne zu wissen, wo dieser im Netz abgelegt ist, so kann er mit den vorhandenen Mitteln dies nicht effizient ermitteln. Suchen, bei welchen nur Teilinhalte von Knoten oder insbesondere auch von externen Dateien bekannt sind, werden nur durch eine Volltextsuche über den gesamten Datenbestand ermöglicht.

## 4.7 Anforderungen an eine Volltextsuche

Die Funktionalität einer Volltextsuche in der FIA lässt sich aufteilen in eine ***Knotensuche*** und eine ***Pfadsuche***. Die *Knotensuche* wird nur kurz beschrieben, da sie analog bekannter Suchmaschinen (z.B. Google [18]) arbeitet. Für die ausführliche Beschreibung der *Pfadsuche* werden zunächst die verwendeten Begriffe definiert.

### 4.7.1 Knotensuche

Die Knotensuche hat zum Ziel, alle Knoten zu ermitteln, deren *semantischer Inhalt* bestimmten Bedingungen erfüllen. Dabei werden alle Knoten getrennt betrachtet und die Struktur des Netzes ignoriert. Diese Form der Suche ist vergleichbar mit der aus dem Internet bekannten Suche über eine Suchmaschine wie Google ([18]), welche Dokumente auf Basis von Bedingungen an deren Inhalte ermittelt.

### 4.7.2 Begriffsdefinitionen für die Pfadsuche

Dieser Abschnitt definiert zunächst für das Verständnis der im nächsten Abschnitt dargestellten Pfadsuche wichtige Begriffe anhand eines Beispielnetzes in Abbildung 4-8.

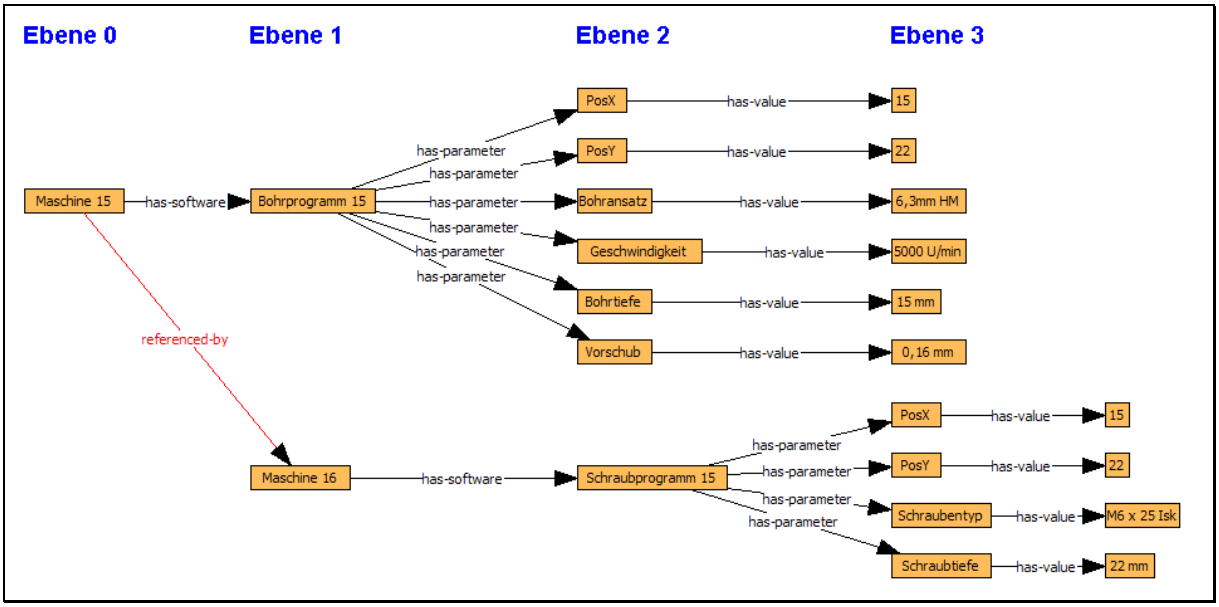


**Abbildung 4-8 Pfade im Netz**

Eine geordnete Liste von Knoten (in der Abbildung durch Rechtecke dargestellt), welche durch Kanten verbunden sind, wird als ***Knotenfolge*** bezeichnet.

Als ***Kantenfolge*** wird eine geordnete Liste von Kanten (in der Abbildung als Pfeile dargestellt) bezeichnet.

Die ***Knotenebene*** bezeichnet ausgehend von einem Startknoten die Entfernung eines weiteren Knotens. Ein Knoten in der Ebene *x* kann vom Startknoten (in der Abbildung "Maschine 15") durch Verfolgung einer *Kantenfolge* der Länge *x* erreicht werden.

Eine geordnete Liste von Kantentypen (die Beschriftung der Pfeile) wird als ***Kantentypfolge*** (auch ***AT-Folge*** genannt) bezeichnet und definiert dabei die semantische Bedeutung der Knoten in den zugehörigen Knotenfolgen. Die *Kantentypfolge* "has-software"→"has-parameter"→"has-value" definiert beispielsweise die semantische Bedeutung der Knoten in der *Knotenebene 3* als Parameterwerte für Software-Programme.

Eine *Knotenfolge* definiert zusammen mit der entsprechenden *Kantentypfolge* einen ***Pfad***. Enthält eine *Kantentypfolge n* Kantentypen (die ***Pfadlänge***), so enthalten die zugehörigen *Knotenfolgen n+1* Knoten. Der letzte Knoten eines Pfades wird als ***Endknoten*** bezeichnet.

Die ***Knotenbedingungen*** definieren Anforderungen an die Inhalte von Knoten einer *Knotenebene*. Dies kann beispielsweise die Existenz eines Wortes sein.

Alle Pfade mit einer bestimmten Kantentypfolge, welche von einem bestimmten Knoten ausgehen werden als ***Pfadkandidaten*** bezeichnet.

### 4.7.3 Pfadsuche

Die Pfadsuche berücksichtigt nicht nur die Knoteninhalte sondern auch die Struktur des Netzes und erlaubt damit gezielte semantische Suchen. Dies soll anhand eines Beispiels erläutert werden. Der Anwender will im Netz aus Abbildung 4-8 alle Knoten ermitteln, welche einen Parameter der "Maschine 15" für eine Position mit dem Wert 15 definieren. Eine Knotensuche nach dem Inhalt "15" würde nicht nur die beiden Parameterwerte 15, sondern auch die drei Knoten "Maschine 15", "Bohrprogramm 15" und "Schraubprogramm 15" zurückgeben. Eine Knotensuche nach dem Teilinhalt "Pos" würde vier Knoten mit den Inhalten "*PosX*" und "*PosY*" zurückgeben. Keine dieser beiden Suchen liefert das gewünschte Ergebnis. Aus diesen Gründen berücksichtigt die Pfadsuche nicht nur verschiedene Knotenbedingungen sondern auch deren semantische Bedeutung. Eine Anfrage wird definiert durch:

- einen Startknoten,
- eine *Kantentypfolge* und
- *Knotenbedingungen* für die jeweiligen *Knotenebenen*.

Die Ergebnismenge der Pfadsuche enthält alle Knoten mit den folgenden Eigenschaften:

- der Knoten ist *Endknoten* eines vom *Startknoten* ausgehenden *Pfadkandidats*
- jeder Knoten auf diesem Pfad entspricht den angegebenen *Knotenbedingungen*.

Tabelle 4-1 zeigt die für eine Pfadsuche notwendigen Suchbedingungen für das o. g. Beispiel.

| **Suchbedingungen** | | |
|---|---|---|
| **Parameter** | **Angabe** | **Bemerkung** |
| Startknoten | "Maschine 15" | |
| Knotenfolge | "has-software"→<br>"has-parameter"→<br>"has-value" | dadurch 4 Ebenen (0-3) für Knoten-bedingungen<br><br>"Maschine 16" wird nicht berücksichtigt, da nur über "referenced-by" erreichbar |
| Knotenbedingungen: | Ebene 2: "Pos*"<br><br>Ebene 3: "15" | passt auf "PosX" und "PosY"<br><br>passt auf Wert von "PosX" und Wert von "Bohrtiefe" |

**Tabelle 4-1 Beispiel für Suchbedingungen**

Das Suchergebnis enthält nun nur den gesuchten Knoten "15", welcher den Wert von "PosX" angibt. Die "Bohrtiefe von 15 mm" ist nicht enthalten, da der Knoten "Bohrtiefe" keine Position angibt. Der Parameter "PosX" mit dem Wert "15" aus Maschine 16 ist nicht enthalten, da dieser nicht über die angegebene Kantentypfolge erreichbar ist (und damit semantisch nicht Bestandteil von Maschine 15 ist).

Das folgende Kapitel 5 beschäftigt sich mit der Umsetzung einer solchen Suchfunktion. Dort werden auch weitere Eigenschaften der Anwendung definiert, sofern sie zum Verständnis der verwendeten Konzepte notwendig sind.

## Kapitel 5

# Suche in semantischen Netzen

Dieses Kapitel beschreibt die Umsetzung der in Kapitel 4 beschriebenen Suchfunktion auf Basis des in Kapitel 3 beschriebenen Frameworks Lucene.

Zunächst werden die ermittelten Grundvarianten in Abschnitt 5.1 beschrieben. Die zusammen mit der Firma mind8 getroffene Auswahl wird in Abschnitt 5.2 erläutert. Die Beschreibung der nachfolgenden Umsetzung in Abschnitt 5.3 konzentriert sich auf die Hauptprobleme der Umsetzung. Kapitel 6 bewertet die Variante durch Tests auf verschiedenen Modellen.

Die Umsetzung der Knotensuche (Abschnitt 4.7.1) ist trivial und wird daher nur kurz in Abschnitt 5.1.1 beschrieben. Die Konzentration in diesem Kapitel liegt auf der Umsetzung der Pfadsuche (Abschnitt 4.7.3).

## *5.1 Varianten der Umsetzung*

Für die Umsetzung der geforderten Pfadsuche existieren drei Grundvarianten mit unterschiedlichen Vor- und Nachteilen. Zunächst erfolgt eine kurze Beschreibung der Knotensuche, da diese zum Teil von der Pfadsuche mitbenutzt wird. Anschließend werden die drei Varianten beschrieben und auf Basis verschiedener Kriterien verglichen.

Für die Pflege der eingesetzten Indizes sind die folgenden Ereignisse relevant:

- Erstellen, Ändern oder Löschen eines Knotens
- Einfügen oder Löschen einer Kante

### 5.1.1 Knotenindex für alle Varianten

Für die Knotensuche wird ein Lucene-Index gepflegt, welcher pro Knoten im Netz ein Dokument enthält. Tabelle 5-1 zeigt den Aufbau eines solchen Dokuments.

| Feldname | Inhalt |
|---|---|
| oid | oid(A) |
| content | content(A) |

**Tabelle 5-1 Dokumente im Knotenindex**

Das Feld *oid* enthält eine eindeutige Referenz auf den Knoten (***OID*** = **O**bject **Id**entifier), welche durch die Funktion ***oid()*** ermittelt wird. Der Inhalt des Knotens, bzw. bei Dateireferenzen der Inhalt der referenzierten Datei, wird durch die Funktion ***content()*** in das Feld content eingefügt.

Dieser Index kann für die Umsetzung der Knotensuche verwendet werden und wird auch benutzt von verschiedenen Varianten der Pfadsuche.

### 5.1.2 Variante 1: "Pfadindex"

Diese Variante ermöglicht eine Pfadsuche durch eine einzige Suchanfrage an Lucene.

Hierzu müssen die für eine Suche verfügbaren *Kantentypfolge* vorher definiert und jeweils ein entsprechender Index erzeugt werden. Die Indizierung umfasst dabei die entsprechenden Pfade inklusive der Knoteninhalte.

Der Inhalt des Index soll an folgendem Beispielnetz für die Folge "at1"→"at2"→"at3" beschrieben werden:

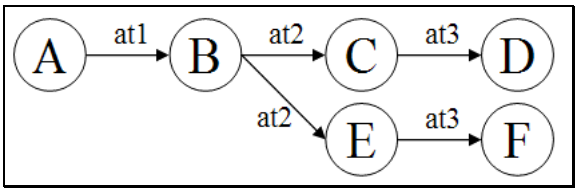


**Abbildung 5-1 Beispielnetz**

Der Index "at1→at2→at3" hat für dieses Teilnetz zwei Dokumente:

| Dokument 1 | |
|---|---|
| **Feldname** | **Inhalt** |
| start-node | oid(A) |
| end-node | oid(D) |
| content0 | content(A) |
| content1 | content(B) |
| content2 | content(C) |
| content3 | content(D) |
| nodes | oid(A).oid(B).oid(C).oid(D) |

| Dokument 2 | |
|---|---|
| **Feldname** | **Inhalt** |
| start-node | oid(A) |
| end-node | oid(F) |
| content0 | content(A) |
| content1 | content(B) |
| content2 | content(E) |
| content3 | content(F) |
| nodes | oid(A).oid(B).oid(E).oid(F) |

**Tabelle 5-2 Dokumente im Pfadindex**

Der Index enthält zu jedem Pfad neben den OIDs auch die Knoteninhalte. Ist ein Knoten in mehreren Pfaden vorhanden, so ist dessen Inhalt auch redundant abgelegt. Änderungen an einem Knoten sind somit relativ teuer, da der Inhalt mehrfach indiziert werden muss.

Das Feld *nodes* wird für die Pfadsuche nicht benötigt. Es wird verwendet um bei Änderungen eines Knotens alle betroffenen Dokumente durch eine Suche nach dessen OID zu ermitteln.

Abbildung 5-2 illustriert eine Pfadsuche in diesem Index.

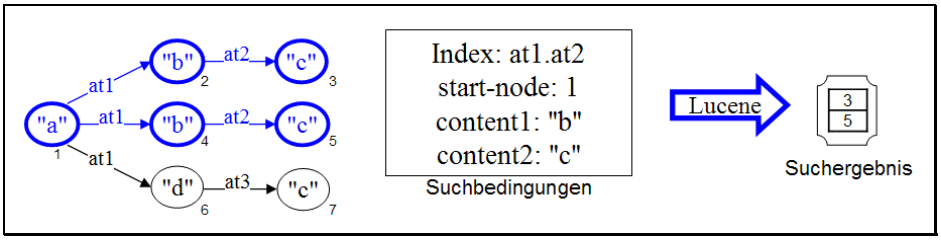


**Abbildung 5-2 Pfad-Suche bei Variante "Pfadindex"**

Die Zahlen unter den Knoten sind die OIDs, der String in den Knoten repräsentiert den Knoteninhalt. Der Index "at1.at2" enthält die fett umrandeten Knoten. Dabei ist Knoten 1 redundant abgelegt. Die Knoten 6 und 7 sind im Index nicht enthalten.

Der Vorteil dieser Variante ist, dass der Index alle Informationen, welche für eine Suche benötigt werden, enthält. Diese Variante bietet damit eine sehr effiziente Suche, erfordert aber durch die redundante Ablage der Knoteninhalte einen hohen Aufwand bei der Indizierung.

### 5.1.3 Variante 2: "Knotenindex"

Diese Variante vermeidet den hohen Indizierungsaufwand der Variante *Pfadindex*, indem nur die Knoteninhalte indiziert werden.

Bei dieser Variante wird nur der Knotenindex (Abschnitt 5.1.1) benötigt. Die Pfadsuche erfolgt durch eine Online-Suche (siehe Abschnitt 2.2.1) über die Kanten und eine indexgestützte Suche (siehe Abschnitt 2.2.2) nach den Knotenbedingungen. Durch Kombination der Ergebnismengen werden die Endknoten für das Suchergebnis ermittelt. Abbildung 5-3 illustriert eine solche Suche.

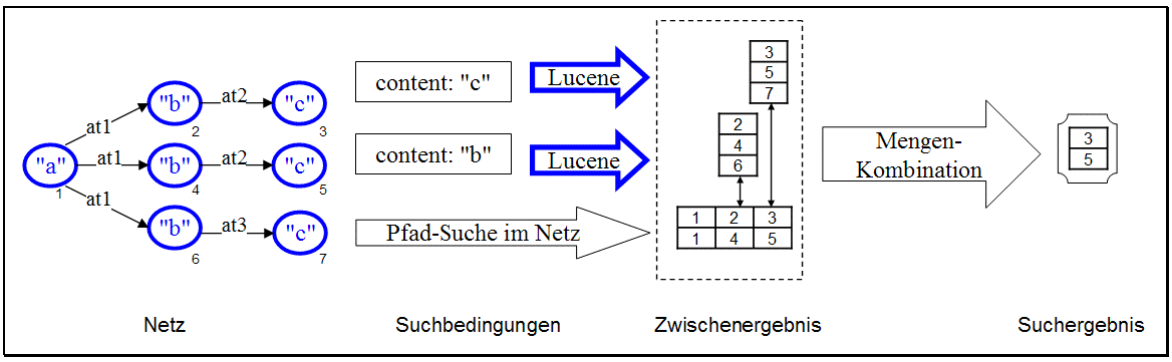


**Abbildung 5-3 Pfad-Suche bei Variante "Knotenindex"**

Eine Lucene-Suche prüft die Knotenbedingungen für jede Knotenebene. Resultat ist hier als Zwischenergebnis jeweils eine Menge von OIDs der entsprechenden Knoten.

Ausgehend vom Startknoten werden alle *Pfadkandidaten* durch eine Online-Suche ermittelt. Im Zwischenergebnis ist dies eine Menge von Knoten-Folgen.

Für die Ermittlung des Suchergebnisses wird nun jede ermittelte Folge mit den Ergebnismengen der Knotensuche verglichen (Mengenkombination). Von allen Pfaden deren Knoten alle in den entsprechenden Knotenmengen enthalten sind, werden die OIDs der Endknoten in das Suchergebnis eingefügt.

Die Indizierung ist bei dieser Variante sehr effizient, da nur der Knotenindex gepflegt werden muss. Die Ereignisse *Kante einfügen* oder *Kante löschen* können ignoriert werden. Des Weiteren müssen die durchsuchbaren Pfade nicht vorher bekannt sein. Die Suche ist dagegen relativ ineffizient, da viele Netzzugriffe (und damit Datenbankzugriffe) zur Ermittlung der Pfade benötigt werden.

### 5.1.4 Variante 3: "Pfad- und Knotenindex"

Diese Variante beschleunigt die Suche gegenüber der Variante *Knotenindex*, erfordert aber nicht die redundante Knotenindizierung der Variante *Pfadindex*. Hierzu werden die beiden Varianten kombiniert. Der Ablauf einer Suche ist ähnlich wie bei der Variante *Knotenindex*, jedoch werden die möglichen Pfade nicht durch eine Online-Suche ermittelt, sondern ähnlich wie bei der Variante *Pfadindex* durch eine Lucene-Suche.

Die Indizierung umfasst analog der Variante "Pfadindex" sowohl die Knoten als auch die AT-Folgen. Ebenso müssen vor der Suche die durchsuchbaren AT-Folgen definiert werden.

Im Gegensatz zur Variante "Pfadindex" werden hier im Pfadindex nicht die Knoteninhalte, sondern nur deren OIDs abgelegt. Bei Knotenänderungen muss also nur der Knotenindex aktualisiert werden.

Tabelle 5-3 zeigt den Inhalt des Pfadindex für die Kantentypfolge "at1"→"at2"→"at3" für das in bereits zuvor verwendete Netz aus Abbildung 5-1.

| Dokument 1 | |
|---|---|
| **Feldname** | **Inhalt** |
| node0 | oid(A) |
| node1 | oid(B) |
| node2 | oid(C) |
| node3 | oid(D) |
| edges | oid(AB) oid(BC) oid(CD) |

| Dokument 2 | |
|---|---|
| **Feldname** | **Inhalt** |
| node0 | oid(A) |
| node1 | oid(B) |
| node2 | oid(E) |
| node3 | oid(F) |
| edges | oid(AB) oid(BE) oid(EF) |

**Tabelle 5-3 Inhalte des Pfadindex at1.at2.at3 (Variante "Pfad- und Knotenindex")**

Der Inhalt der Felder *"nodeX"* definiert die Zugehörigkeit zur entsprechenden Knotenebene. Das Feld *"edges"* enthält die OIDs der Kanten und wird verwendet um bei Löschungen von Kanten die betroffenen Dokumente über eine Lucene-Suche zu ermitteln.

Abbildung 5-4 zeigt den Ablauf einer Pfadsuche bei dieser Variante.

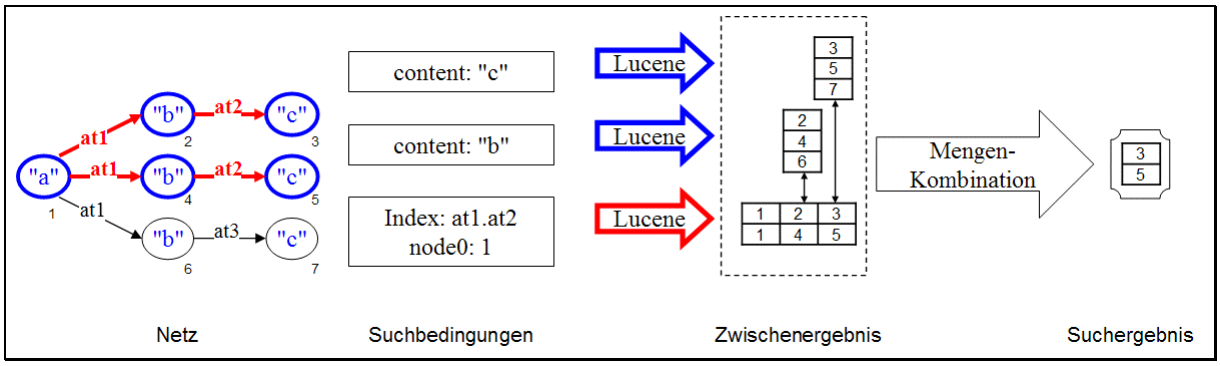


**Abbildung 5-4 Pfadsuche bei Variante "Pfad- und Knotenindex"**

Im Gegensatz zur Variante *Knotenindex* werden hier auch die *Pfadkandidaten* über eine Lucene-Suche ermittelt.

Der Vorteil dieser Variante gegenüber der Variante *Pfadindex* ist, dass eine Suche ohne Netzzugriffe erfolgen kann, auch wenn dafür eine Indizierung der Kantenfolgen notwendig

ist. Wird bei einer Suche der Startknoten nicht angegeben so kann auch eine Suche über alle vorhandenen *Pfade* der angegebenen *Kantentypfolge* erfolgen.

Für die Pflege dieser Indexstruktur illustriert Abbildung 5-5 exemplarisch die nötige Reaktion auf das Ereignis "Neue Kante mit Typ *AT3* von *A* nach *B* angefügt".

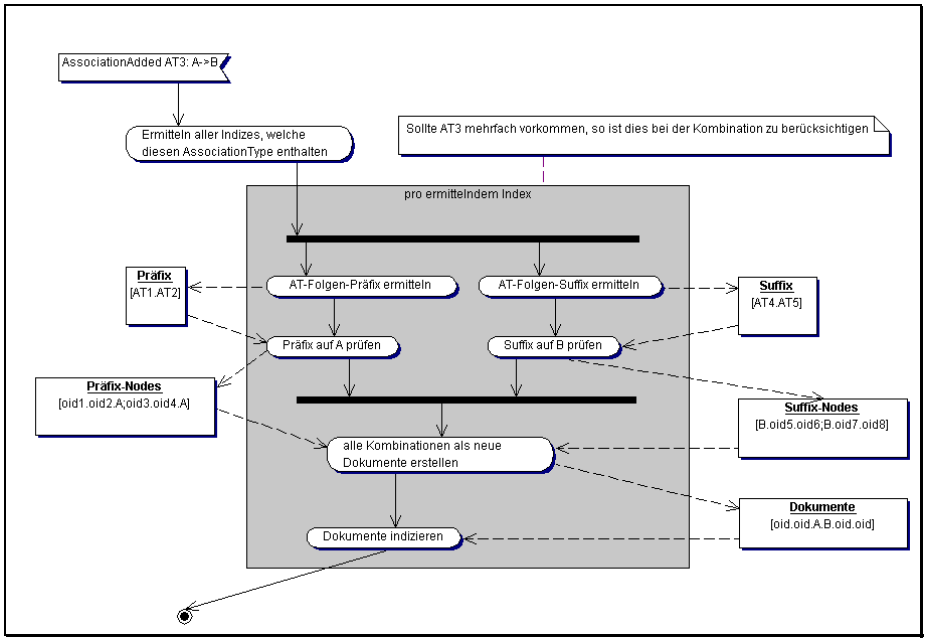


**Abbildung 5-5 Ablauf beim Einfügen einer neuen Kante**

Im ersten Schritt werden basierend auf dem Typ der neuen Kante (in der Abbildung *"AT3"*) alle Indizes ermittelt, welche diese Kante enthalten könnten. Anschließend werden für jeden Index die Präfix und die Suffix-Teilpfade ermittelt. Dies wird in Abbildung 5-6 illustriert.

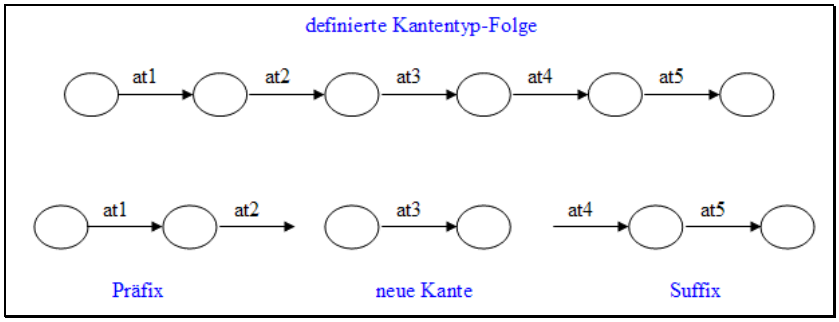


**Abbildung 5-6 Präfix und Suffix**

Nach der Ermittlung des Präfix und Suffix wird geprüft, ob eine dem Präfix entsprechende Folge am Startknoten der neuen Kante endet (Präfix-Folge) und ob eine dem Suffix entsprechende Folge vom Endknoten der neuen Kante ausgeht (Suffix-Folge). Durch dieses Verfahren können mehrere Präfix-und Suffix-Folgen entstehen. Alle Kombinationen "Präfix-Folge" - "neue Kante" - "Suffix-Folge" werden jeweils als ein Dokument in den Index eingefügt.

### 5.1.5 Vergleich der Varianten

Tabelle 5-4 vergleicht die drei Varianten aufgrund von wichtigen Effizienzkriterien. Tabelle 5-5 nennt weitere Kriterien, aus welchen sich die Effizienzbetrachtungen weitestgehend ableiten.

| **Kriterium** | **Pfadindex** | **Knotenindex** | **Pfad- und Knotenindex** |
|---|---|---|---|
| Zeit-Effizienz der Pfadsuche | ++ | – | + |
| Zeit-Effizienz der Indizierung bei Knotenänderungen | – | + | + |
| Zeit-Effizienz der Indizierung bei Strukturänderungen | – | ++ | + |
| Platzeffizienz der Index-Ablage | – | ++ | + |
| Zeiteffizienz der Indizierung steigt mit Anzahl definierter Kantentyp-Folgen? | – – | nein | – |
| **Legende** | | | |
| + = gut | | – = schlecht | |

**Tabelle 5-4 Effizienzvergleich der drei Varianten**

| **Kriterium** | **Pfadindex** | **Knotenindex** | **Pfad- und Knotenindex** |
|---|---|---|---|
| Suche benötigt Netzzugriffe? | - | 👎 | - |
| Redundante Ablage der Knoteninhalte im Index? | 👎 | - | - |
| Vordefinition durchsuchbarer Kantentyp-Folgen erforderlich? | 👎 | - | 👎 |
| Aussagekräftige Relevanz-Angabe ohne aufwändige Zusatzmaßnahmen möglich? | ✓ | - | - |
| Suche über alle Pfade möglich | ✓ | - | ✓ |
| **Legende** | | | |
| 👎 = ja (negativ) | ✓ = ja (positiv) | - = nein | |

**Tabelle 5-5 Vergleich weiterer Eigenschaften der Varianten**

Berücksichtigt werden hier nur die Effizienzkriterien, welche während der Ausführung der Anwendung relevant sind. Die Effizienz der notwendigen Index-Erstellung wird hier nicht berücksichtigt, da sie einmalige Ereignisse betrifft.

Eine aussagekräftige Relevanz-Angabe ist nur bei der Variante Pfadindex möglich, da nur diese die Pfadsuche durch einen einzigen Index-Zugriff realisiert.

Die Variante *Pfadindex* schneidet im Vergleich deutlich am Schlechtesten ab. Sie ist aber die einzige Variante, welche Relevanz-Angaben in den Suchergebnissen liefert.

Die Variante *Knotenindex* spart Indizierungszeit auf Kosten der für eine Pfadsuche benötigten Zeit. Dafür kann sie durchgeführt werden, ohne dass für die entsprechenden Pfade zuvor Indizes angelegt werden müssen. Sie eignet sich demnach für adHoc-Suchen.

Die Variante *Pfad- und Knotenindex* vereint die meisten Vorteile der beiden anderen Varianten. Sie hat eine bessere Such-Effizienz, verringert dafür aber die Indizierungseffizienz. Diese Variante eignet sich insbesondere für oft wiederkehrende Suchanfragen.

## 5.2 Auswahl einer Variante

Die Auswahl einer Variante erfolgte im Rahmen eines Treffens mit den Entwicklern der mind8 GmbH.

Die Variante "*Pfad- und Knotenindex*" erschien als guter Kompromiss zwischen Indizierungs- und Sucheffizienz und wurde daher für die Umsetzung ausgewählt. Für Suchanfragen, welche nicht auf einen vorhandenen Index zurückgreifen wird die Variante "Knotenindex" realisiert. Dies erlaubt einen empirischen Vergleich der beiden Varianten und überlässt die Wahl ob ein Index verwendet und gepflegt wird der Anwendung bzw. dem Anwender.

## 5.3 Umsetzung

Dieser Abschnitt beschreibt die Umsetzung der Anforderungen in der FIA. Zunächst wird ein Überblick über das neu erstellte Eclipse-Plugin gegeben. Die Schnittstelle zur Anwendung wird dabei auf einer implementierungsnahen Ebene vorgestellt. Im Weiteren werden einzelne Bereiche auf einer konzeptionellen Ebene beschrieben.

### 5.3.1 Aufbau der Komponente

Abbildung 5-7 zeigt eine Übersicht über die Hauptbestandteile der Implementierung.

Die Komponente zerfällt in die drei Unterkomponenten *Schnittstelle*, *Lucene-Zugriff* und *Suche*.

Die *Schnittstelle* wird in Abschnitt 5.3.2 beschrieben. Der *Lucene-Zugriff* wird durch *Processor-Klassen* gekapselt, welche die einzelnen Prozesse implementieren. Für die beiden Index-Arten existieren spezialisierte Klassen, welche lediglich den Lucene-Zugriff implementieren. Dies erlaubt den Austausch von Lucene durch ein anderes Suchframework ohne größere Änderungen. Die Komponente *Suche* implementiert die in den Abschnitten 5.1.3 und 5.1.4 beschriebenen Such-Verfahren und greift dabei auf die *Processor-Klassen* zurück.

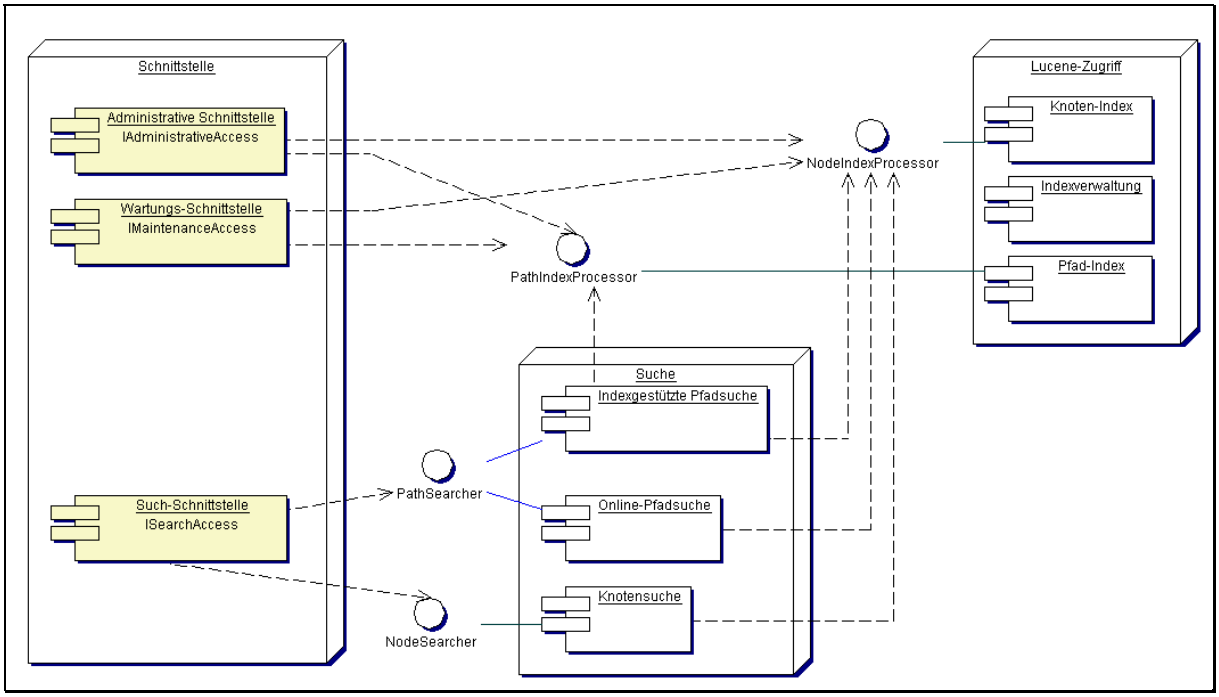


**Abbildung 5-7 Grober Aufbau der Implementierung**

## 5.3.2 Schnittstelle zur Anwendung

Abbildung 5-8 zeigt die Anbindung der Volltextsuche an die bestehende Anwendung.

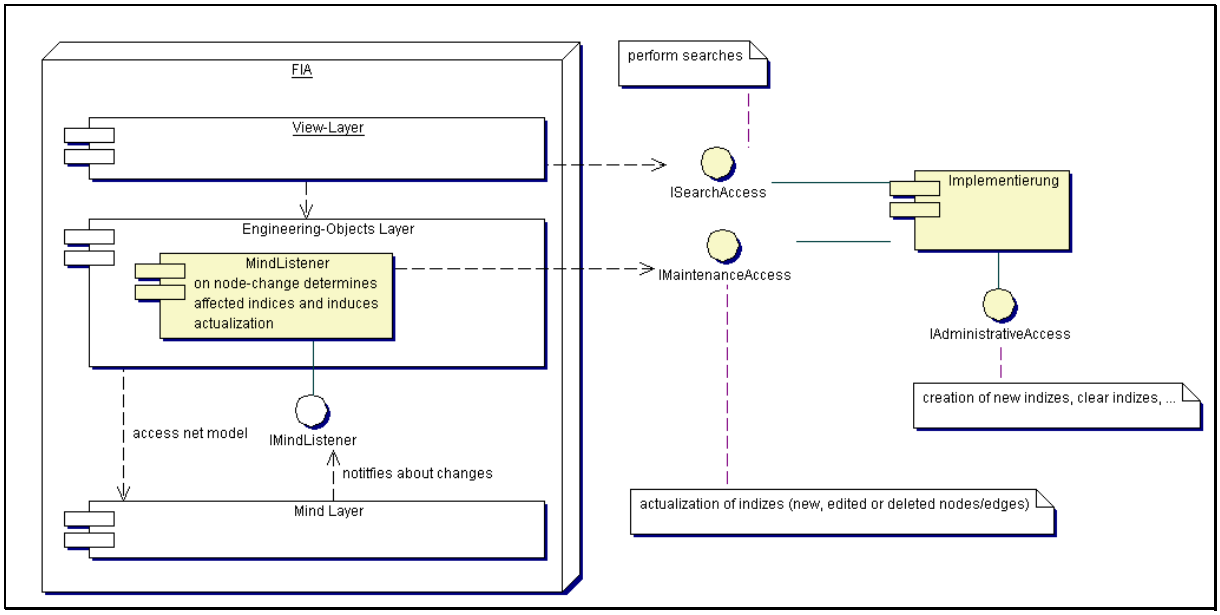


**Abbildung 5-8 Anbindung der Suche an die FIA**

Die neuen Komponenten sind in der Abbildung gelb hinterlegt. Diese werden hier nun kurz vorgestellt und anschließend vertieft. Der MindListener dient der Aktualisierung der Indizes basierend auf Änderungen im Netz und bedient die ***Wartungs-Schnittstelle*** IMaintenanceAccess. Diese wird in Abschnitt 5.3.4 beschrieben. Die ***Such-Schnittstelle*** ISearchAccess dient der Durchführung von Suchen. Über die ***Administrative Schnittstelle***

IAdministrativeAccess können die Indizes verwaltet werden. Der Zugriff auf die Implementierungen dieser Schnittstellen erfolgt über eine Singleton-Factory-Klasse.

Abbildung 5-9 zeigt die Schnittstellen für die Index-Verwaltung und Suche IAdministrativeAccess und ISearchAccess (in der Abbildung gelb hinterlegt) zusammen mit den nötigen Typ-Definitionen.

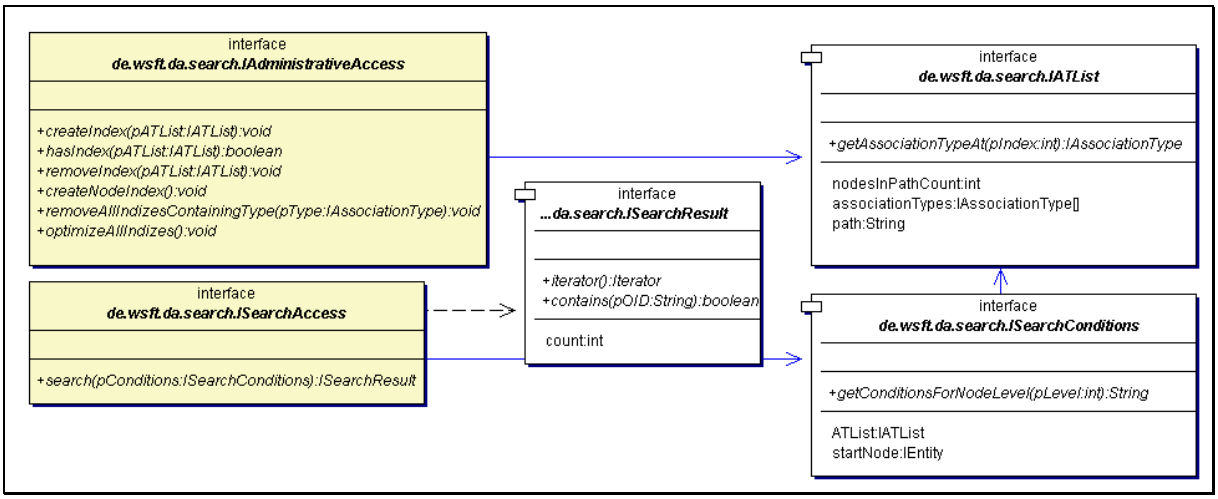


**Abbildung 5-9 Schnittstelle der Komponente zur Volltextsuche**

Die Schnittstelle IAdministrativeAccess bietet Methoden zur Anlage, Existenzabfrage und Löschung von Pfadindizes und zur Neuindizierung des Knotenindex. Zusätzlich bietet sie eine Methode zur Optimierung aller Indizes (siehe Abschnitt 5.3.9). Ein Index wird über eine Instanz von IATList, welche eine *Kantentypfolge* definiert, identifiziert.

Das Interface ISearchAccess bietet eine einzelne Methode `search`, welche Suchen auf dem Netz erlaubt. Die Suchbedingungen werden in Form einer Instanz von ISearchConditions als Parameter spezifiziert.

ISearchConditions enthält verschiedene Eigenschaften (Properties), welche die Form der Suche festlegen. Wird `ATList` mit einem gültigen Wert belegt, so erfolgt eine Pfadsuche, bei *null* erfolgt eine Knotensuche. Die Bedingungen an die Knoten der einzelnen Ebenen werden über die Methode `getConditionsForNodeLevel` zurückgegeben. Ist `startNode` belegt, so erfolgt eine Pfadsuche ausgehend von diesem Knoten, bei *null* erfolgt eine Suche über alle vorhandenen Pfade des angegeben Typs, jedoch muss hierfür ein Index vorhanden sein. Die Pfadsuche erfolgt indexgestützt (Variante Pfad- und Knotenindex, Abschnitt 5.1.4) oder online (Variante Knotenindex, Abschnitt 5.1.3), je nachdem ob ein entsprechender Index vorhanden ist. Ungültige Suchanfragen werden über den Java-Exception-Mechanismus dem Aufrufer gemeldet.

### 5.3.3 Erstellung eines Pfadindex

Die initiale Erstellung eines Pfadindex erfolgt durch eine Traversierung aller Knoten im Netz. Abbildung 5-10 zeigt exemplarisch den Ablauf bei der Anlage eines neuen Pfadindex.

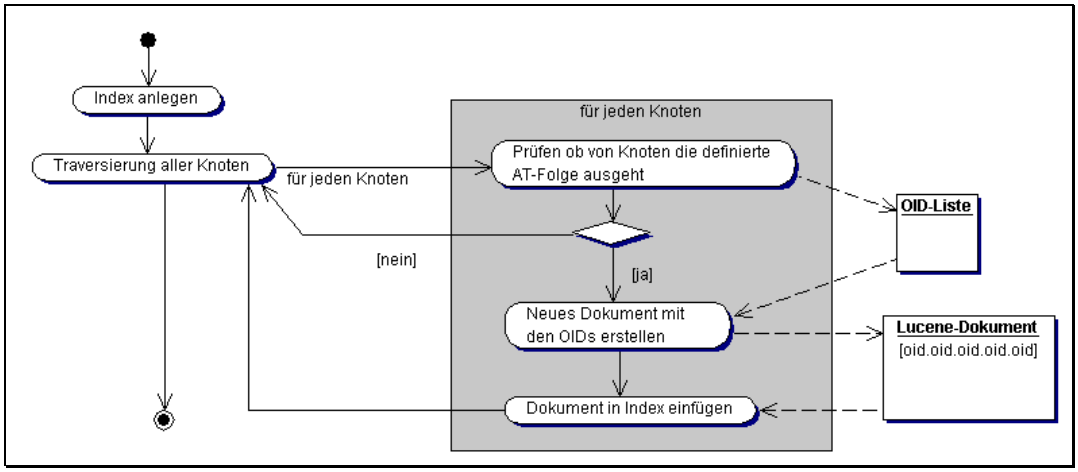


**Abbildung 5-10 Ablauf bei der Anlage eines neuen Pfadindex**

Das Verfahren nutzt einen BruteForce-Ansatz. Hierbei wird für jeden im Netz existierenden Knoten geprüft, ob von diesem Pfade der entsprechenden Kantentypfolge ausgehen. Da die Neu-Anlage eines Index ein relativ seltenes Ereignis ist, wurde hier gemäß dem Prinzip "Keep it simple" keine aufwändige Optimierung des Algorithmus vorgenommen. Ein alternativer Algorithmus könnte basierend auf dem ersten Kantentyp der Kantentypfolge alle Kandidaten für Startknoten auswählen und von dort aus die vorhandenen Pfade ermitteln. Dieses Verfahren verspricht eine hohe Effizienz insbesondere dann, wenn die relative Anzahl solcher Kanten gering ist.

### 5.3.4 Änderungen im Netz

Änderungen im Netz beinhalten die Anlage, Bearbeitung und Löschung von Knoten, sowie die Anlage und Löschung von Kanten. Als Reaktion müssen die entsprechenden Lucene-Indizes aktualisiert werden.

Die FIA definiert eine Schnittstelle für derartige Reaktionen in Abbildung 5-11.

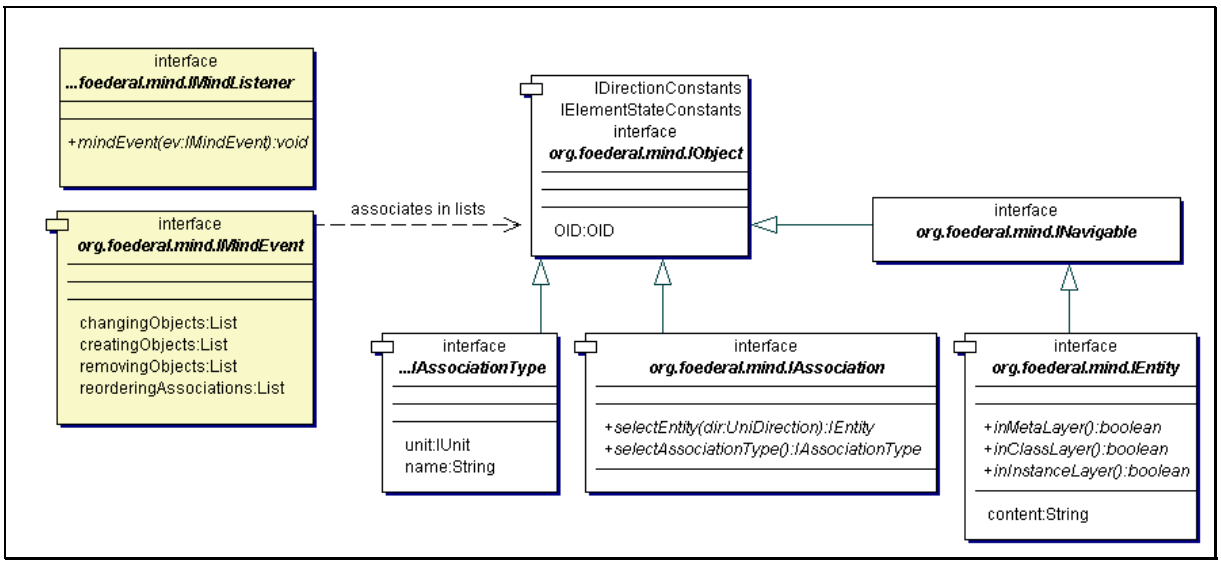


**Abbildung 5-11 Listener für Reaktion auf Netzänderungen**

Kern der Schnittstelle sind die in der Abbildung gelb hinterlegten Klassen IMindListener und IMindEvent.

Die Klasse IMindListener folgt dem Observer-Pattern ([16, p.293ff]) und baut auf einem Event-Mechanismus auf. Implementierungen dieses Interface können sich in der FIA registrieren und werden dann bei Änderungen im Netz durch Aufruf der Methode `mindEvent` benachrichtigt. Die als Parameter übergebene Instanz von IMindEvent enthält die Änderungen im Netz. Diese werden durch Listen, welche Instanzen von IObject enthalten, dargestellt. Das Interface IObject ist ein Super-Interface für die im Rahmen dieser Arbeit relevanten Objekte Knoten (IEntity), Kante (IAssociation) und KantenTyp (IAssociationType).

Die Änderungen werden klassifiziert nach

- neu erzeugte Objekte (`creatingObjects`)
- geänderte Objekte (`changingObjects`)
- gelöschte Objekte (`removingObjects`)

Die Speicherung von Änderungen in der Anwendung erfolgen in der Datenbank unter Verwendung eines Two-Phase-Commit-Protokolls ([21], p. 395ff). Der MindListener wird dabei direkt vor dem Commit über alle in dieser Transaktion vorgenommenen Änderungen benachrichtigt. Die Menge der Objekte variiert stark je nach Anwendungsfall. Wird beispielsweise über den Modelleditor ein Knoten eingefügt oder bearbeitet, so löst dies bereits eine solche Transaktion aus. Die Duplizierung vieler Knoten dagegen erfolgt auch in einer einzelnen Transaktion.

Die Such-Komponente muss auf diese Änderungen reagieren und die entsprechenden Lucene-Indizes aktualisieren. Die Komponente stellt hierfür die in Abbildung 5-12 dargestellte Schnittstelle IMaintenanceAccess bereit, welche durch eine Implementierung von IMindListener bedient wird.

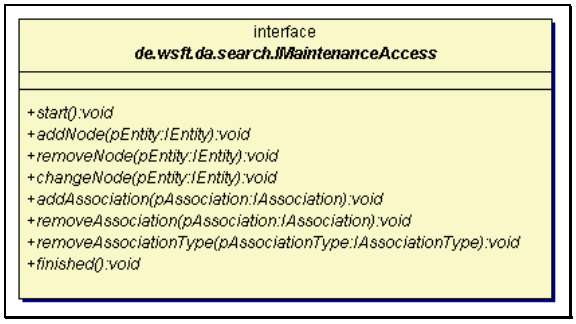


**Abbildung 5-12 Schnittstelle zur Pflege der Indizes**

Die Verarbeitung eines IMindEvent wird durch die Methoden `start` und `finished` umschlossen. Der Zweck dieser beiden Methoden wird in Abschnitt 5.3.7 beschrieben. Dazwischen werden nacheinander die einzelnen Änderungen an die entsprechenden Methoden des IMaintenanceAccess gemeldet. Tabelle 5-6 nennt die notwendigen Reaktionen auf diese Ereignisse.

| Ereignis | Reaktion |
|---|---|
| `addNode` | Einfügen des Knotens in den Knotenindex |
| `removeNode` | Entfernen des Knotens aus dem Knotenindex und Reindizierung aller erbenden Knoten (siehe Abschnitt 5.3.5) |
| `changeNode` | Aktualisierung des Knotens im Knotenindex und Aktualisierung aller erbenden Knoten (siehe Abschnitt 5.3.5) |
| `addAssociation` | Ermitteln aller Pfadindizes, welche eine Kante dieses Typs enthalten können und jeweils Prüfung ob durch diese Kante ein neuer Pfad erzeugt wird. Bei Bedarf in die entsprechenden Indizes neue Pfade einfügen |
| `removeAssociation` | Löschen aller Pfade in allen Pfadindizes, welche diese Kante enthalten. |
| `removeAssociationType` | Löschen aller Pfadindizes, welche diesen Kantentyp enthalten. |

**Tabelle 5-6 Reaktionen auf Netz-Ereignisse**

### 5.3.5 Vererbung von Knoteninhalten

Da die Suche nach semantischen Aspekten erfolgt muss die in Abschnitt 4.5 beschriebene Vererbung der Knoteninhalte bei der Indizierung berücksichtigt werden, so dass diese für die Suche transparent ist. Als Konsequenz muss bei der Änderung des Inhalts eines Knotens nicht nur dieser Knoten im Index aktualisiert werden, sondern alle Knoten, welche diesen Knoten benutzen. Abbildung 5-13 verdeutlicht dies anhand vier aufeinander folgender Änderungsszenarien.

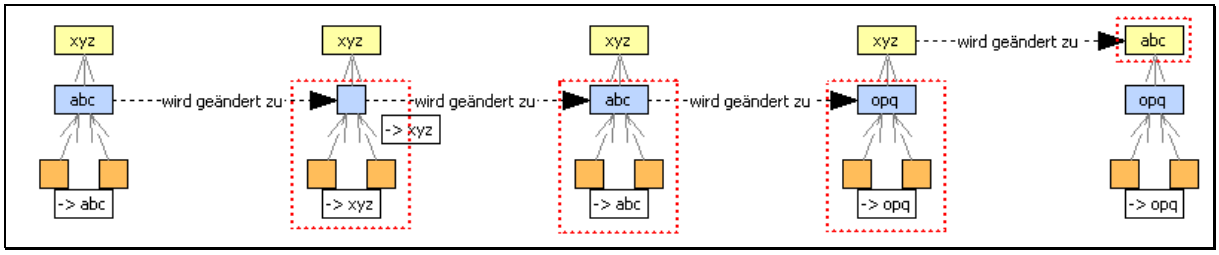


**Abbildung 5-13 Auswirkungen von Knoten-Inhalts-Änderung bei Vererbung**

Die Abbildung zeigt die vier möglichen Szenarien für Änderungen von Knoteninhalten. Die horizontalen Pfeile definieren die vorgenommenen Änderungen. Die von der entsprechenden Änderung betroffenen Knoten sind jeweils rot gestrichelt umrandet. Die zu diesen Knoten zugehörigen Lucene-Dokumente müssen im Index aktualisiert werden. In den ersten drei Fällen müssen die Lucene-Dokumente der beiden (orangenen) *Instanz*-Knoten und des (blauen) *Class*-Knotens aktualisiert werden, da diese semantisch von der Änderung betroffen sind. Im letzten Fall muss nur das Lucene-Dokument des *Meta*-Knotens aktualisiert werden, da der Class-Knoten dessen Inhalt überschreibt.

Änderungen an der Instanziierungs-Hierarchie müssen ebenfalls beachtet werden. Abbildung 5-14 illustriert dies anhand zweier aufeinander folgenden Änderungen.

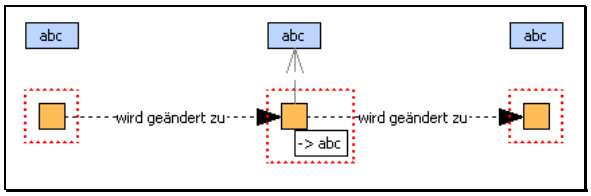


**Abbildung 5-14 Auswirkungen von Instanziierungs-Änderungen bei Vererbung**

Wie die Abbildung zeigt, müssen bei Änderungen der Instanziierungsbeziehungen alle Dokumente der erbenden Knoten aktualisiert werden. Im ersten und letzten Fall ist für den (orangenen) *Instanz*-Knoten semantisch kein Inhalt vorhanden. Für die Suche müsste dieser im Index also nicht vorhanden sein. Aus Implementierungsgründen wird er trotzdem im Index abgelegt (s. u.).

Eine effiziente Implementierung setzt voraus, dass alle zu ändernden Knoten schnell aufgefunden werden können. Dies wird erreicht, indem die Dokumente im Knotenindex um ein zusätzliches Feld "inheritance" erweitert werden. Dieses Feld listet die OIDs aller Knoten auf den verwendeten Vererbungspfaden auf. Um bei Hierarchie-Änderungen ebenfalls über eine Lucene-Suche die betroffenen Knoten ermitteln zu können, werden auch für Knoten welche semantisch keinen Inhalt haben entsprechende Dokumente angelegt. Abbildung 5-15 zeigt dies anhand von Beispielen.

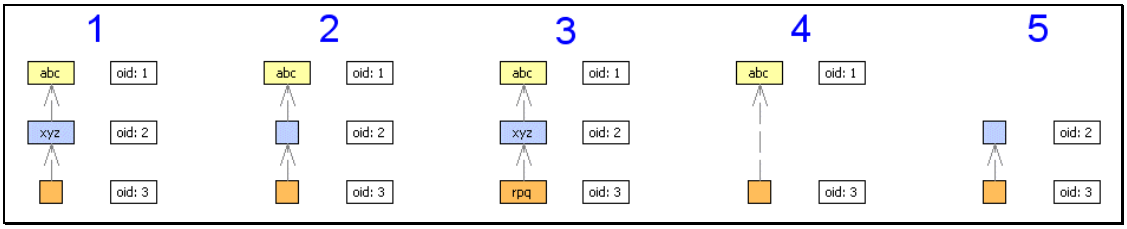


**Abbildung 5-15 Beispiele für Vererbungshierarchien**

Die Inhalte der zugehörigen Lucene-Dokumente für die Instanz-Knoten zeigt Tabelle 5-7.

| Feld | Beispiel 1 | Beispiel 2 | Beispiel 3 | Beispiel 4 | Beispiel 5 |
|---|---|---|---|---|---|
| oid | 3 | 3 | 3 | 3 | 3 |
| content | xyz | abc | rpq | abc | |
| inheritance | 3 2 | 3 2 1 | 3 | 3 1 | 3 2 |

**Tabelle 5-7 Lucene-Dokumente für Instanz-Knoten aus Abbildung 5-15**

Bei Änderungen von Knoteninhalten können alle betroffenen Knoten durch eine Suche nach der OID des geänderten Knotens im Feld *inheritance* aufgefunden werden. Bei Änderungen der Instanziierungsbeziehungen ist dies über eine Suche nach der OID des erbenden Knotens möglich.

Bei der Indizierung müssen sowohl die Vererbung von Knoteninhalten als auch Verweise auf externe Dateien berücksichtigt werden. Abbildung 5-16 zeigt den Ablauf bei der Ermittlung des semantischen Inhalts eines Knotens.

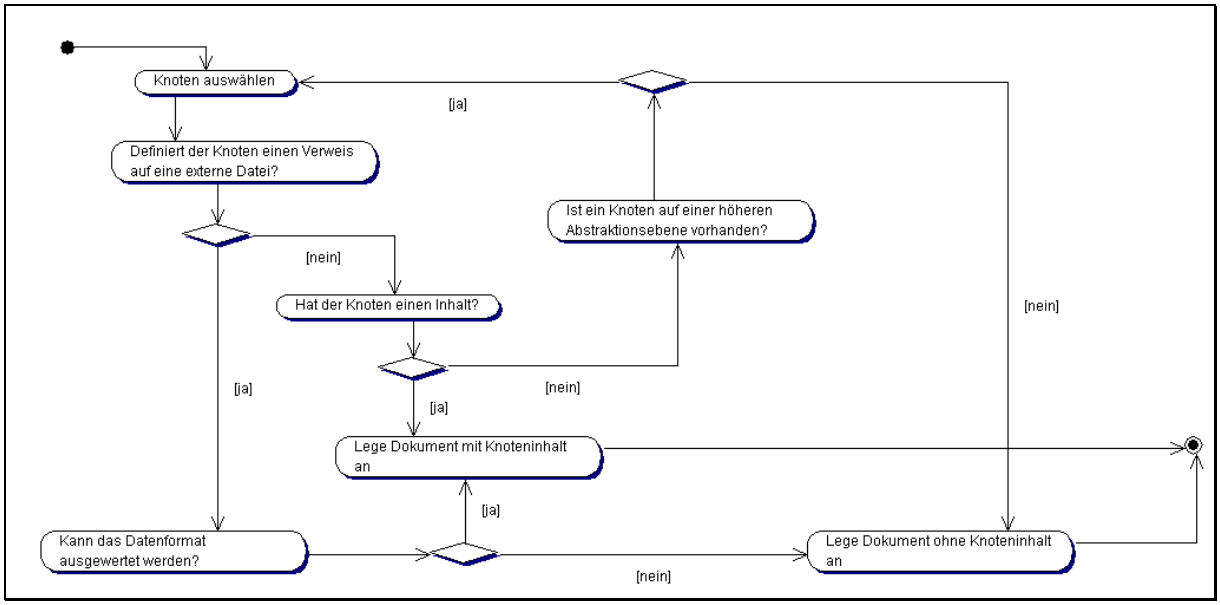


**Abbildung 5-16 Ermittlung des semantischen Inhalts eines Knotens**

Zu beachten ist hier, dass die Prüfung auf eine externe Referenz vor der Prüfung eines leeren Inhalts erfolgt. Dies hat den Hintergrund, dass die Ermittlung von Referenzen im Modell nicht einheitlich ist, sondern je nach Teilnetz unterschiedlich erfolgt. Es ist denkbar, dass ein leerer Inhalt gerade eine solche Referenz definiert, weswegen die Reihenfolge der Prüfungen relevant ist.

### 5.3.6 Indexverwaltung

Da für Pfade mehrere Indizes existieren können, müssen die Inhalte und Ablageorte verwaltet werden. Die in Abbildung 5-17 dargestellte IndexLocator-Komponente übernimmt diese Verwaltung.

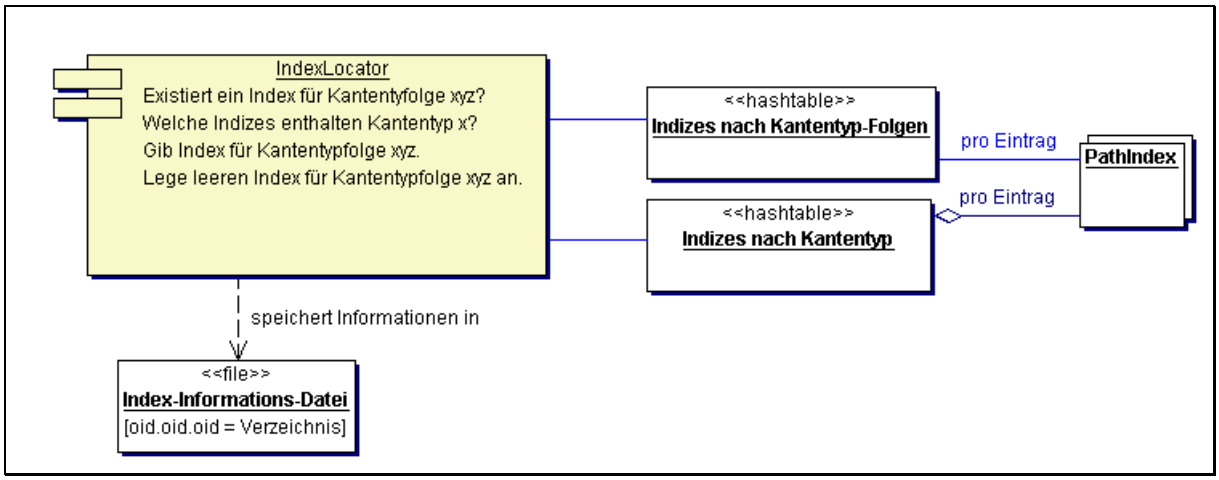


**Abbildung 5-17 Indexverwaltung**

Die relevanten Aufgaben der Komponente sind in der Abbildung eingetragen. Im Wesentlichen besteht die Komponente aus zwei Hashtables. Die Hashtable *"Indizes nach Kantentyp-Folgen"* verbindet jede definierte Kantentypfolge mit der entsprechenden Index-Instanz und erlaubt damit die Ermittlung eines Index basierend auf der definierten

Kantentypfolge. Zu diesem Zweck wird die Folge als String durch Konkatenierung der Kantentyp-OIDs dargestellt.

Die Hashtable "Indizes nach Kantentyp" verbindet einzelne Kantentypen mit einer Liste der Indizes, welche diesen Typ enthalten. Dies erlaubt bei der Reaktion auf Änderungen im Netz eine effiziente Ermittlung der eventuell betroffenen Indizes (vgl. Abbildung 5-5).

Die Index-Informations-Datei dient der persistenten Ablage von Informationen zu den definierten Indizes und erlaubt eine effiziente Initialisierung des IndexLocators. Implementiert ist sie als eine properties-Datei, welche die String-Darstellung der Kantentypfolgen mit den entsprechenden Verzeichnissen auf dem Dateisystem verbindet. Die Informationen in den Hashtables können aus diesen Informationen vollständig ermittelt werden.

### 5.3.7 Vermeidung von Mehrfach-Indizierung

Die Auslösung von MindEvents baut auf dem verwendeten Transaktions-Konzept auf. Das bedeutet, dass in einem Event nicht nur einzelne Objekte enthalten sein können, sondern auch sehr viele. Die Duplizierung eines Teilnetzes erfolgt beispielsweise in einer einzigen Transaktion. Das entsprechende MindEvent enthält dementsprechend ganze Pfade. Die Pfade sind im Event als einzelne Kanten dargestellt. Abbildung 5-18 zeigt ein fiktives Teilnetz.

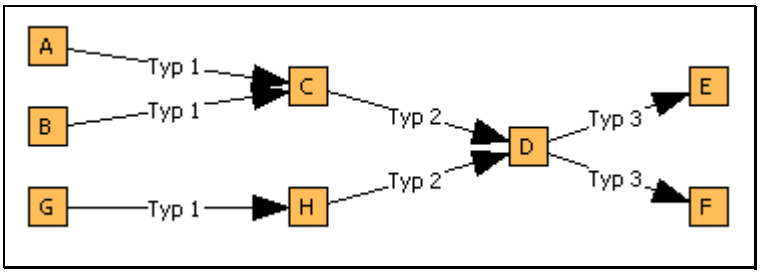


**Abbildung 5-18 Duplizierung eines Teilnetzes**

Wird dieses Netz dupliziert, so wird ein MindEvent ausgelöst, welches 8 Knoten und 7 Kanten enthält. In den Index "Typ 1 – Typ 2 – Typ 3" müssen 6 Pfade (z.B. A-C-D-E) eingefügt werden. Der in Abschnitt 5.1.4, Abbildung 5-5 dargestellte Algorithmus fügt allerdings 18 neue Pfade in den Index ein. Dies liegt daran, dass für jede neue Kante geprüft wird, welche Pfade mit dieser Kante erzeugt werden können.

Würde jede Kante in einer eigenen Transaktion und damit in einem separaten MindEvent angelegt, so wäre dieses Problem nicht vorhanden, da erst die letzte Kante einen entsprechenden Pfad erzeugt.

Eine Änderung des MindEvent-Konzepts ist zur Lösung dieses Problems allerdings nicht notwendig. Stattdessen wird eine Menge gepflegt, welche alle neu eingefügten Pfade enthält. Vor dem Einfügen in den Index wird geprüft, ob der entsprechende Pfad in dieser Menge enthalten ist. Ist dies nicht der Fall, so wird der Pfad in den Index und die Menge eingefügt. Die Menge ist als HashSet realisiert und erlaubt so eine effiziente Existenzprüfung. Die Pfade werden als String durch Konkatenierung der Knoten- und Kanten-OIDs dargestellt.

Die Lebensdauer dieser Menge muss auf eine Unit-of-Work beschränkt werden. Aus diesem Grund wird die Menge beim Aufruf der Methode `start` erzeugt und beim Aufruf von `finish` wieder gelöscht.

### 5.3.8 Unterstützung verschiedener Datenformate

Daten in externen Dateien können verschiedene Formate haben. Dokumentationen können beispielsweise in Form von XML oder RTF vorliegen. Steuerungsprogramme liegen dagegen in einer bestimmten Programmiersprache, wie AWL vor.

Für den Aufbau der Lucene-Dokumente müssen aus diesen Dateien die relevanten Daten extrahiert und die einzelnen Wörter in das *Dokument* getrennt durch Leerzeichen eingefügt werden. Die Definition des Begriffs *Wort* hängt dabei vom Format ab. In einem Text ist ein Wort gemäß der üblichen Definition eine Reihe von Zeichen und Ziffern ohne Leer- und Satzzeichen. In dem Java-Kommentar dagegen "//Dies ist ein Kommentar" sind die Zeichen "//" nicht Bestandteil des ersten Wortes und müssen eliminiert werden. Der Algorithmus, welcher diese Extraktion vornimmt, benutzt einen Parser für die syntaktische Analyse. Der Algorithmus welcher zusätzlich den extrahierten Text in ein Lucene-Dokument einfügt wird hier ***TextParser*** genannt.

In vielen Suchmaschinen werden die einzelnen Bestandteile einer Datei typisiert (z.B. nach Autor, Titel etc.) um hier spezifischere Suchanfragen zu erlauben. Im Rahmen der FIA ist diese Typisierung nicht erforderlich und aufgrund der verschiedenen Textformen (Text, Variablenname, ...) auch nicht sinnvoll einsetzbar.

Die FIA klassifiziert die Datenformate gemäß dem von der IETF definierten Mime-System ([25]). Bei der Indizierung externer Dateien muss nun zunächst der zum jeweiligen Typ passende Parser gefunden werden. Abbildung 5-19 zeigt die Ermittlung eines Parsers durch die Klasse ParserLocator.

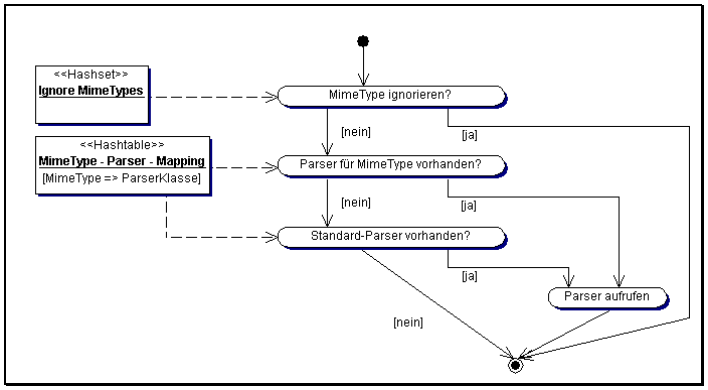


**Abbildung 5-19 Ermittlung eines Parsers für einen gegebenen MimeType**

Im ersten Schritt wird geprüft, ob dieser MimeType ignoriert werden soll. Dies erfolgt durch eine Existenzprüfung in einer Menge (*Ignore-Menge*). Ist der MimeType in dieser Menge enthalten, so wird der Inhalt der externen Datei ignoriert. Im zweiten Schritt wird nun über eine Hashtable ein passender Parser gesucht. Diese Hashtable verwendet den MimeType als Key und enthält den Namen der zu verwendenden Parser-Klasse, bzw. eine entsprechende Instanz als Wert. Ist hier kein Eintrag für den MimeType vorhanden, so wird nach einem Eintrag mit dem MimeType "*" gesucht. Dies dient als Standard für alle nicht explizit

definierten Typen. Wird kein Standard-Parser definiert, so müssen alle zu berücksichtigenden MimeTypes in der Hashtable definiert werden. Die *Ignore-Menge* ist in diesem Fall nicht notwendig, da alle nicht definierten MimeTypes ignoriert werden.

Beim ersten Zugriff auf den ParserLocator werden die entsprechenden Parser-Klassen mit ihrem *Fully Qualified Name* (FQN, siehe Anhang A ) in die Hashtable eingetragen. Die Hashtable enthält also zunächst nur Strings als Werte. Wird nun ein entsprechender Parser benötigt, so wird eine Instanz erzeugt und der Wert in der Hashtable durch diese ersetzt. Dieses Vorgehen bewirkt, dass Instanzen nicht unnötig erzeugt werden.

Abbildung 5-20 zeigt die Klasse ParserLocator und die Basisklassen für *TextParser*.

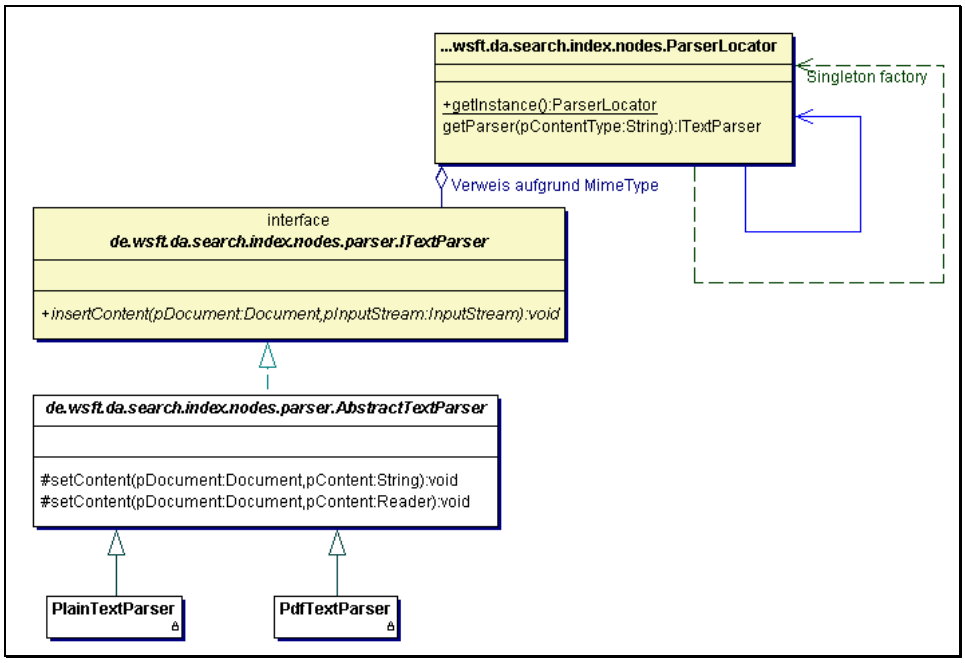


**Abbildung 5-20 Parser für verschiedene Datenformate**

Jeder Parser muss das Interface ITextParser implementieren und einen öffentlichen Konstruktor ohne Parameter (Leer-Konstruktor, Default-Konstruktor) bereitstellen. Die Erzeugung eines Lucene-Dokumentes und die Definition aller Felder, mit Ausnahme des Inhalts, wird durch die bestehende Implementierung vorgenommen. Die Implementierung des Parsers erhält als Parameter das Lucene-Dokument sowie einen InputStream, welcher den Inhalt der externen Datei enthält. Der Parser muss nur noch über eine der in der Klasse AbstractTextParser vorhandenen Methoden `setContent` den Inhalt entsprechend in das Dokument einfügen.

### 5.3.9 Index-Optimierung

Die Optimierung der Indizes bringt Performance-Vorteile für die Suche (vgl. Abschnitt 3.5.3). Der Aufwand für die Optimierung steigt dabei mit der Indexgröße, da alle Daten des Index in neue Dateien zusammengefasst werden. Insbesondere wenn der zuvor optimierte Index im Vergleich zu den Änderungen groß ist, kann hier der Zeitaufwand bedeutsam sein. Da MindEvents bei manueller Bedienung der Anwendung beim Erzeugen jedes einzelnen Knotens und jeder einzelnen Kante auftreten, wird eine Optimierung sehr oft ausgeführt.

Viele Änderungen am Netz verursachen eine Fragmentierung des Index und verringern damit die Such-Performance. Die in Abschnitt 3.5.3 beschriebene automatische Segment-Zusammenführung ist dadurch allerdings nicht betroffen, so dass trotzdem periodisch eine Optimierung erfolgt. Diese Optimierung ist allerdings in den seltensten Fällen vollständig, da diese, wie in Abschnitt 3.5.3 beschrieben nur dann erfolgt, wenn die Anzahl der vorliegenden Segmente eine Potenz von 10 ist.

Eine manuelle Optimierung kann durch die administrative Schnittstelle IAdministrativeAccess (siehe Abschnitt 5.3.2) vorgenommen werden. Durch die Wahl des Zeitpunkts und der Häufigkeit dieser Optimierung muss ein Kompromiss zwischen Such-Effizienz und flüssigem Arbeiten gefunden werden. Da bereits optimierte Indizes durch Lucene nicht unnötigerweise erneut optimiert werden, ist ein zu häufiges Aufrufen kein Problem. Sinnvoll erscheinende Optimierungszeitpunkte sind der Anwendungsstart, regelmäßige Intervalle (z.B. täglich) oder auch direkt vor Beginn einer Suche. Zu beachten ist, dass auch die Verarbeitung von MindEvents im Fall von gelöschten oder geänderten Objekten die Suche nutzt und damit deren Effizienz durch einen nicht-optimierten Index negativ beeinflusst wird.

In jedem Fall sollte dem Anwender zusätzlich die Möglichkeit einer manuellen Optimierung gegeben werden, um durch die Anwendung nicht definierbare Arbeitsabläufe zu unterstützen. Wird beispielsweise ein großes Modell neu erstellt so ist eine manuelle Optimierung nach Abschluss der Erstellung sinnvoll.

Zu Beachten ist allerdings, dass eine zu seltene Optimierung zu Laufzeitfehlern aufgrund zu vieler geöffneter Dateien führen kann. Eine Migration auf die kommende Lucene-Version 1.3 wird dieses Problem etwas entschärfen, da dort die Anzahl der Dateien eines Segmentes gegenüber der eingesetzten Version 1.2 verringert wurde (siehe Abschnitt 3.5.4).

Die aktuelle Implementierung nimmt eine Optimierung nach jedem MindEvent vor, um die oben genannten Probleme zu vermeiden und außerdem einen Bug in der Version 1.2 zu umgehen. Dieser Bug verursacht eine Zerstörung des Index, wenn bestimmte Anzahlen von Dokumenten erzeugt und vor der nächsten Optimierung wieder entfernt werden.

## Kapitel 6

# Analyse der Umsetzung

Dieses Kapitel analysiert die in Kapitel 5 beschriebene Implementierung durch empirische Messung der Laufzeiten verschiedener Anwendungsfälle auf verschiedenen Netzen, welche durch den in Abschnitt 6.1 beschriebenen Modellgenerators erzeugt wurden. Abschnitt 6.2 definiert die für die Messungen verwendeten Anwendungsfälle. Die Testumgebung und das Testverfahren wird in Abschnitt 6.3 beschrieben, deren Durchfügung in Abschnitt 6.4. Die Ergebnisse der Messungen werden in Abschnitt 6.5 vorgestellt. Abschnitt 6.6 bewertet die Implementierung auf Basis der Messungen.

Als Kriterium für eine Bewertung der Konzepte wird die benötigte Laufzeit verwendet. Diese wird durch Delta-Bildung zwischen den Systemzeiten bei Start- und Ende gemessen. Dieses Verfahren wurde gewählt, da es die für den Anwender sichtbare Zeit widerspiegelt. Relevant sind nicht nur die Laufzeiten der einzelnen Anwendungsfälle auf dem bestehenden Modell, sondern auch Voraussagen über deren Entwicklung mit steigender Netzgröße.

## *6.1 Modellgenerator*

Um Messungen auf verschiedenen Netzgrößen vornehmen zu können, wurde ein Modell-Generator entwickelt. Dieser erzeugt eine beliebige Anzahl von Pfaden mit beliebiger Länge. Dabei ist die Struktur des Netzes durch die Angabe von Wahrscheinlichkeiten für neue Kanten beeinflussbar. Die Knoteninhalte werden durch Textdateien definiert, aus welchen jeweils eine beliebige Zeile verwendet wird.

Abbildung 6-1 zeigt ein Beispiel für ein generiertes Modell.

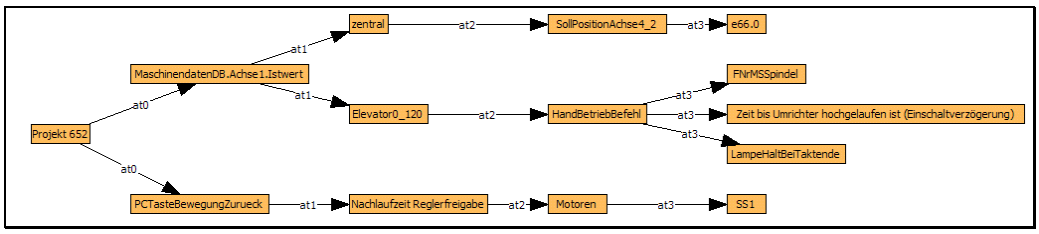


**Abbildung 6-1 Generiertes Modell**

Code-Snippet 6-1 zeigt die Steuerungsanweisungen, welche zur Generierung dieses Modells verwendet wurden.

```
1:   File tProjectName = new File("projectNames.txt");
2:   File tFile = new File("contents.txt");
3:   Model tModel = new Model("model-B");
4:   tModel.createNewPath(5).
5:        createStartNode(tProjectName, 1000, 0.2).
6:        addEdge("has-subProject", 0.1).
7:        addNode(tFile, 5000).
8:        addEdge("at1", 0.2).
9:        addNode(tFile, 5000).
10:       addEdge("at2", 0.1).
11:       addNode(tFile, 5000).
12:       addEdge("at3", 1.0).
13:       addNode(tFile, 5000);
14:  tModel.createNewPath(10).
15:       ...
16:  ModelGenerator.createModel(tModel);
```

**Code-Snippet 6-1 Steuerungsanweisungen für den Modellgenerator**

Die Zeilen 1 und 2 definieren Textdateien für die Knoteninhalte. In Zeile 3 wird ein neues Modell definiert. Der Parameter *"model-B"* definiert die organisatorische Einheit (Unit) in welcher das Modell generiert werden soll. Dies erlaubt die Generierung mehrerer Modelle in einer Datenbank. Durch die Im- und Export-Funktionen der FIA, welche auf einzelnen Units arbeiten können die Modelle später wieder getrennt werden. Ist diese Unit bei der Generierung bereits vorhanden, so werden deren Inhalte gelöscht. Zeile 4 fügt eine neue Pfaddefinition an das Modell an. Der Parameter definiert die Anzahl der zu generierenden Pfade. In Zeile 5 werden die Startknoten für diesen Pfad definiert. Dabei wird im ersten Parameter die Textdatei für die Knoteninhalte angegeben. Der zweite Parameter gibt an, dass nur die ersten 1000 Zeilen der Datei verwendet werden sollen. Dies erlaubt eine Beeinflussung der Streuung der Werte und erlaubt gleichzeitig die Verwendung derselben Dateien für Modelle verschiedener Größen. Der dritte Parameter gibt schließlich die Wahrscheinlichkeit an, mit welcher ein neuer Knoten erzeugt werden soll. Wird ein bestehender Knoten verwendet, so wird dieser zufällig aus den vorhandenen Startknoten ausgewählt. Die Zeile 6 definiert eine vom Startknoten ausgehende Kante. Der erste Parameter definiert dabei den Kantentyp in Form eines Strings. Die Kantentypen werden global für alle Modelle in einer eigenen Unit definiert. Der zweite Parameter definiert die Wahrscheinlichkeit mit welcher eine neue Kante erzeugt wird. Analog zum Startknoten wird auch hier bei bestehenden Kanten zufällig eine ausgewählt. Zeile 7 fügt, falls eine neue Kante erzeugt wird, einen Knoten an diese Kante an. Die Parameter definieren hierbei analog zum Startknoten wieder eine Textdatei und eine Begrenzung der beachteten Zeilen in dieser Datei. Die Zeilen 8-13 arbeiten analog den Zeilen 6 und 7. Ein Pfad muss mit einer Knotendefinition abgeschlossen werden. Um genau die definierte Anzahl der Pfade zu erhalten muss die letzte Kante immer neu erzeugt werden (Wahrscheinlichkeit = 1.0). In Zeile 14 wird eine weitere Pfaddefinition angefügt, deren Auswirkung in der Abbildung 6-1 allerdings nicht dargestellt sind. Zeile 16 startet schließlich die Generierung des Modells.

Um für die Messungen sowohl eine definierte Pfad- als auch Knotenanzahl zu erhalten wird bei allen Kanten (Zeilen 6, 8, 10 und 12) eine Wahrscheinlichkeit von 1.0 angegeben und nur eine einzelne Pfaddefinition verwendet. Abbildung 6-2 zeigt das Ergebnis dieser Änderung.

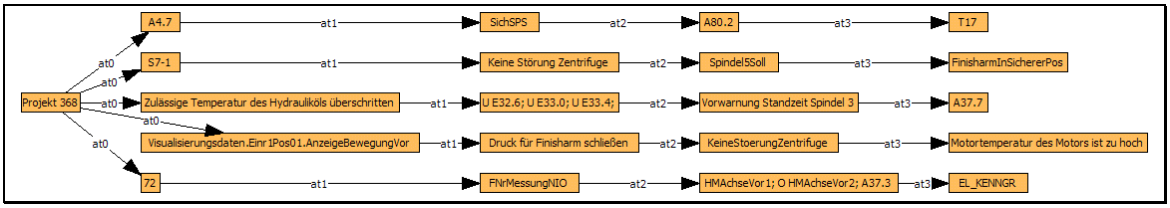


**Abbildung 6-2 Modellstruktur für Messungen**

Die Anzahl der Knoten lässt sich nun nach der Formel $\#Knoten = \#Pfade * Pfadlänge + 1$ berechnen.

## 6.2 Anwendungsfälle für die Messungen

Die Messungen werden anhand verschiedener Szenarios durchgeführt, welche auch ***Anwendungsfälle*** genannt werden. Diese werden analog der Aufspaltung der Schnittstellen klassifiziert in die drei Gruppen *Administration*, *Suche* und *Indexpflege*.

Die folgende Aufstellung nennt die bei den Messungen berücksichtigten Anwendungsfälle. Für spätere Referenzen sind diese mit eindeutigen Namen gekennzeichnet. Jeder Anwendungsfall wird mit verschiedenen Datenmengen (z.B. Netzgröße) ausgeführt um hier Prognosen für die Entwicklung in der Zukunft vornehmen zu können.

- Administration:
  - Erstellen des Knotenindex (*CreateNodeIndex*):
    - ►Knotenanzahl: 10.000, 20.000, ..., 100.000, 250.000
  - Erstellen eines Pfadindex (*CreatePathIndex*):
    - ►Pfadlänge: 2, 3, ..., 10
    - ►Pfadanzahl: 1.000, 2.000, ..., 10.000, 25.000
- Suche:
  - Online-Pfadsuche (*OnlineSearch*):
    - ►Pfadlänge: 2, 3, ..., 10
    - ►Pfadanzahl: 10, 20, ..., 100, 250, 500
  - Index-Pfadsuche (*IndexSearch*):
    - ►Pfadlänge: 2, 3, ..., 10
    - ►Pfadanzahl: 10, 20, ..., 100, 250, 500
- Indexpflege:
  - Erstellung neuer Knoten (*CreateNodes*):
    - ►Knotenanzahl: 1, 5, 10, 20, ..., 100
  - Erstellung neuer Pfade auf bestehenden Knoten (*CreatePaths*):
    - ►Pfadlänge: 2, 5
    - ►betroffene Indizes: 1, 2, ..., 10
    - ►Pfadanzahl: 10, 20, ..., 100, 250, 500

Für *CreateNodeIndex* und *CreatePathIndex* wurden Modelle (***CreateIndices-Modelle***) mit 1.000, 2.000, ..., 10.000 und 25.000 Pfaden der Länge 10 erstellt. Jedes dieser Modelle wurde durch einen Export als eigenständige Datei abgelegt, so dass die Messungen jeweils auf einer neu angelegten Datenbank nach einem Import erfolgen können. Dies vermeidet Einflüsse durch Fragmentierungen in der Datenbank. Das bestehende reale Modell enthält 15.106 Knoten und liegt damit in der Größe zwischen den beiden kleinsten generierten Modellen. Im Gegensatz zum realen Modell verwenden die generierten Modelle keine externen Referenzen sondern legen alle Daten direkt in den Knoten ab. Dies vermeidet anwendungsexterne Einflüsse durch das Dateisystem. Ursprünglich sollte auch ein Modell mit 50.000 Pfaden für die Messungen verwendet werden. Diese Modellgröße führte aber beim Import zu extremen Skalierungsproblemen mit der bestehenden Anwendung und wurde daher für den Test nicht berücksichtigt.

Für *OnlineSearch* und *IndexSearch* wurde ein einzelnes Modell (***Suchmodell***) generiert, welches mehrere Pfadmengen enthält. Alle *x* Pfade einer Pfadmenge haben die Länge 10 und gehen von einem dedizierten Knoten (dem jeweiligen Such-Startknoten) aus. Das Modell besteht aus den Pfadmengen für $x$ = 10, 20, ..., 100, 250 und 500. Jeder Knoten einer Knotenebene hat denselben Inhalt, wodurch die Angabe von Knotenbedingungen auf jeder Ebene möglich ist, ohne die Ergebnismenge dabei einzuschränken. Dies stellt den WorstCase der Suche dar, da in jeder Knotenebene ein Mengenvergleich stattfindet. Durch die 12 *Such-Startknoten* können gezielt die einzelnen Pfadmengen angesprochen werden.

Abbildung 6-3 verdeutlicht die Struktur des Suchmodells.

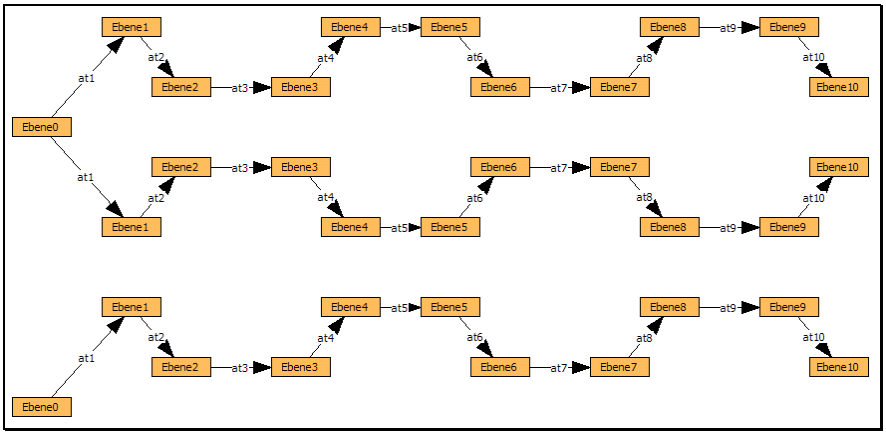


**Abbildung 6-3 Struktur des Suchmodells**

Die Abbildung zeigt die Pfadmenge für x=1 und x=2.

Die Modelle für *CreateNodes* and *CreatePaths* müssen programmatisch zur Laufzeit erstellt werden, da sie der Bewertung dynamischer Aspekte der Implementierung dienen.

## 6.3 Testumgebung und Testverfahren

Als Testumgebung wird ein PC mit einem Intel Pentium IV 1,8 GHz und 768 MB Hauptspeicher unter Windows 2000 verwendet. Für die Datenablage kommt die objektorientierte Datenbank Poet FastObjects ([32]) in der Version 9.0 zum Einsatz. Ein durch die Datenbank optional verfügbares Objekt-Caching wird nicht vorgenommen, um nicht-nachvollziehbare Einflüsse auf die Messwerte zu begrenzen.

Bei allen Messungen werden die Laufzeiten nach den Bereichen ***Netzzugriffe***, ***Lucene-Zugriffe*** und ***Lucene-Anbindung*** aufgeteilt. *Netzzugriffe* beinhaltet dabei alle Zugriffe auf die bestehende Anwendung (z.B. Abfrage der OID eines Knotens). Unter *Lucene-Zugriffe* fallen alle Zugriffe auf die Indizes (Erstellung, Suche, Einfügen, Löschen). Die *Lucene-Anbindung* umfasst die restlichen Aufwände. Diese bestehen aus den im Rahmen dieser Arbeit implementierten Algorithmen.

Die Laufzeiten der Anwendungsfälle *OnlineSearch* und *IndexSearch* sind intuitiv unabhängig von der Gesamtgröße des vorhandenen Netzes. Dies begründet sich auf den folgenden Annahmen:

1. Der Zeitbedarf für Lesezugriffe unterscheidet sich in einem großen Netz nur unwesentlich gegenüber einem kleinen Netz.

2. Die Index-Suche erfolgt weitgehend im Hauptspeicher, so dass auch hier der Einfluss der Netzgröße vernachlässigt werden kann.

Aus diesem Grund werden die Messungen nicht auf vielen verschiedenen Netzgrößen durchgeführt. Um allerdings auch nicht realitätsfern nur auf dem Suchmodell zu operieren erfolgen die Messungen auf der Vereinigung des *Suchmodells* mit dem *CreateIndices-Modell* mit 2.000 Pfaden. Durch eine Vergleichsmessung auf der Vereinigung des Suchmodells mit dem *CreateIndices-Modell* mit 10.000 Knoten wurden die Annahmen bestätigt. *IndexSearch* ergab exakt denselben Gesamtaufwand. Bei *OnlineSearch* (Pfadlänge 6) ergab sich eine Differenz des Gesamtaufwandes von 5 ms (ca. 0,08 %)

Auch die Laufzeiten der Anwendungsfälle *CreateNodes* und *CreatePaths* sind (ohne Optimierung) intuitiv unabhängig von der Gesamtgröße des vorhandenen Netzes. Dies begründet sich hier auf den folgenden Annahmen:

1. Der Zeitbedarf für Einfüge-Operationen unterscheidet sich in einem großen Netz nur unwesentlich gegenüber einem kleinen Netz.

2. Ohne Optimierung werden nur neue Dokumente in neuen Segmenten erzeugt, wodurch kein Einfluss der bestehenden Indexgröße vorhanden ist.

Diese Anwendungsfälle werden auf dem *CreateIndices-Modell* mit 2.000 Pfaden ausgeführt. Eine Vergleichsmessung auf dem *CreateIndices-Modell* mit 10.000 Pfaden bestätigte auch hier die Annahmen. Die Abweichung ohne Optimierung betrug beispielsweise bei *CreateNodes* in der Summe 570 ms (etwa 6 %), welche hauptsächlich den Lucene-Zugriff betrifft. Der Grund liegt vermutlich in einem etwas schlechteren I/O-Verhalten aufgrund der größeren Speicherbelastung.

Vor Beginn einer Messreihe wird eine neue Datenbank erstellt, die entsprechenden Modelle importiert und die Anwendung neu gestartet. Dies führt zu einem reproduzierbaren Anwendungsstatus für die Messungen.

Alle Messungen werden dreifach vorgenommen und die Durchschnittswerte ermittelt. Dies minimiert den Einfluss punktueller externer Einflüsse (z.B. des Betriebssystems). Aufgrund der niedrigen Zeitwerte bei der Indexpflege werden die Messungen fünffach durchgeführt und offensichtliche extreme Ausreißer ignoriert.

Direkt vor der Ermittlung der Messwerte wird der entsprechende Anwendungsfall zwei bis dreimal ohne Messung ausgeführt, um einen realitätsnahen Zustand der Anwendung zu erreichen. Zusätzlich wird direkt vor Beginn eines Anwendungsfalls der GarbageCollector explizit aufgerufen, um hier Einflüsse durch den nicht-deterministischen GarbageCollector-Thread auf die Messwerte zu verringern. Dieses Verfahren bewirkte im Test eine Annäherung der zuvor teilweise extrem variierenden Messwerte. Dies ist dadurch zu erklären, dass Initialisierungen einzelner Klassen (z.B. Aufbau der Datenbank-Verbindung, Einlesen von Konfigurationsdateien etc.) nur beim ersten Aufruf erfolgen. Da in der Regel die Anwendung bereits eine Zeitlang läuft, wenn die Suchfunktionen verwendet werden, sind diese Initialisierungen bereits erfolgt, so dass diese in die Messwerte nicht einfließen sollten. Die Mehrfach-Ausführung bewirkt auch Initialisierungen, welche erst beim zweiten oder dritten Mal ausgeführt werden (z.B. Online-Optimierung der Datenbank-Verbindung).

## 6.4 Ermittlung der Messwerte

Die Anwendungsfälle *CreateNodeIndex* und *CreatePathIndex* werden gemeinsam in einer Messreihe pro *CreateIndices-Modell* gemessen. Hierzu wird jeweils zunächst *CreateNodeIndex* gefolgt von *CreatePathIndex* mit den verschiedenen Pfadlängen ausgeführt.

Eine weitere Messreihe umfasst die Anwendungsfälle *OnlineSearch* und *IndexSearch*. Hier wird zunächst für jeden *Such-Startknoten* nacheinander *OnlineSearch* mit den verschiedenen Pfadlängen ausgeführt. Anschließend werden für die verschiedenen Pfadlängen Indizes erstellt (ohne Messung) und die Suchen als *IndexSearch* wiederholt.

Die letzte Messreihe umfasst die Anwendungsfälle *CreateNodes* und *CreatePaths*. Zunächst wird *CreateNodes* mit den verschiedenen Knotenzahlen ausgeführt. Da die Messungen mehrfach erfolgen, werden die Knoten jeweils anschließend ohne Messung gelöscht. Als nächstes wird *CreatePaths* für die einzelnen Pfadlängen ausgeführt. Hierzu werden jedes Mal die benötigten Knoten ohne Messung erstellt und anschließend die entsprechenden Kanten eingefügt. Auch hier werden die Kanten nach jeder Messung gelöscht.

## 6.5 Testergebnisse

Dieser Abschnitt fasst die Ergebnisse der Messungen zusammen. Für alle Messungen genügte der vorhandene Hauptspeicher, so dass keine Einflüsse durch Auslagerung (Swapping) vorhanden sind. Als erstes werden die Anwendungsfälle im Bereich Administration (CreateNodeIndex und CreatePathIndex) beschrieben. Es folgen die Messungen der Suche (OnlineSearch und IndexSearch) und der Index-Pflege (CreateNodes und CreatePaths).

### 6.5.1 Administration

Abbildung 6-4 zeigt die Ergebnisse der Messungen für den Anwendungsfall *CreateNodeIndex*.

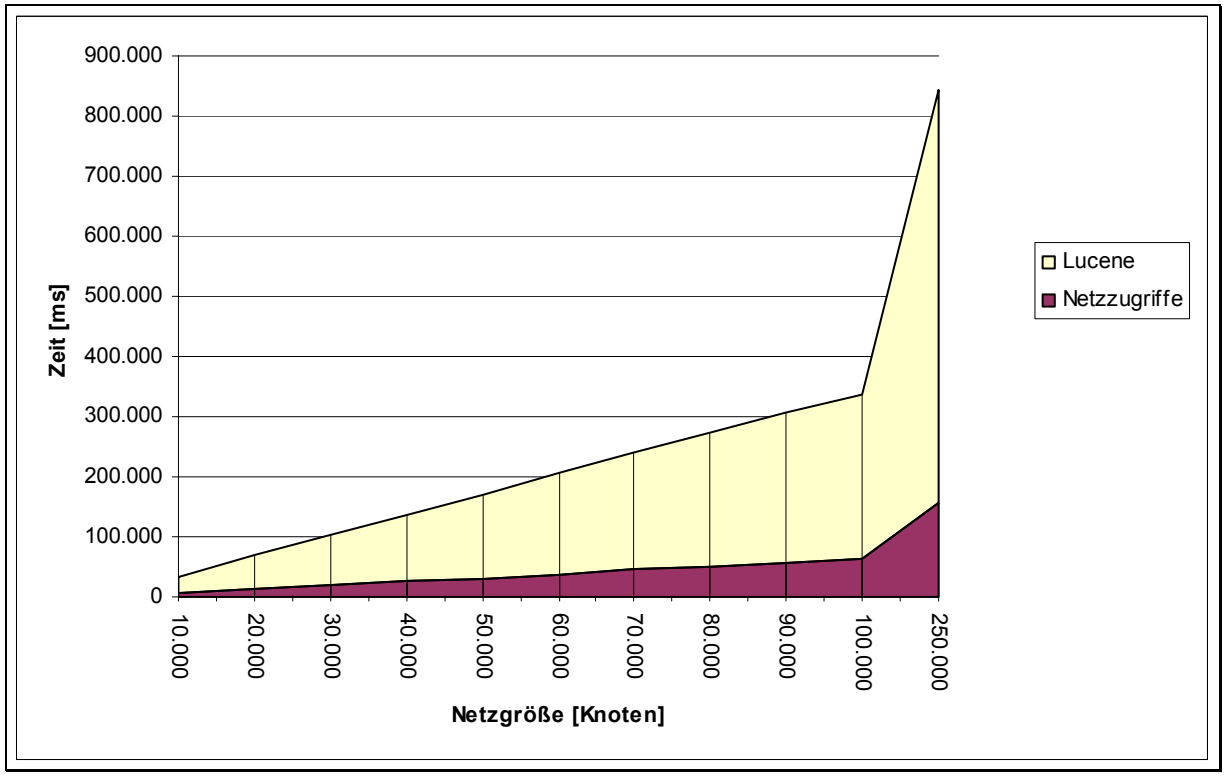


**Abbildung 6-4 Messwerte für den Anwendungsfall CreateNodeIndex**

Im Diagramm werden die Aufwände für die *Lucene-Anbindung* und die *Netz-Zugriffe* zusammengefasst, da die Lucene-Anbindung einen kaum darstellbaren Anteil ausmacht (0,5% - 1,2 %). Das Diagramm zeigt eine lineare Abhängigkeit zwischen dem Zeitaufwand und der Netzgröße auf, wobei die durchschnittliche Zeit pro Knoten zwischen 3,4 und 3,5 ms schwankt. Der Anteil der Netz-Zugriffe weist eine geringe Steigerung mit der Netzgröße auf, welche durch vermutlich durch die steigende Größe der Datenbank verursacht wird. Bei 10.000 Knoten beträgt der Anteil 17,8% und bei 250.000 Knoten 18,7%.

Die Messwerte für den Anwendungsfall *CreatePathIndex* werden in mehreren Diagrammen dargestellt, da hier zwei Variablen (Pfadlänge und Pfadanzahl) berücksichtigt werden.

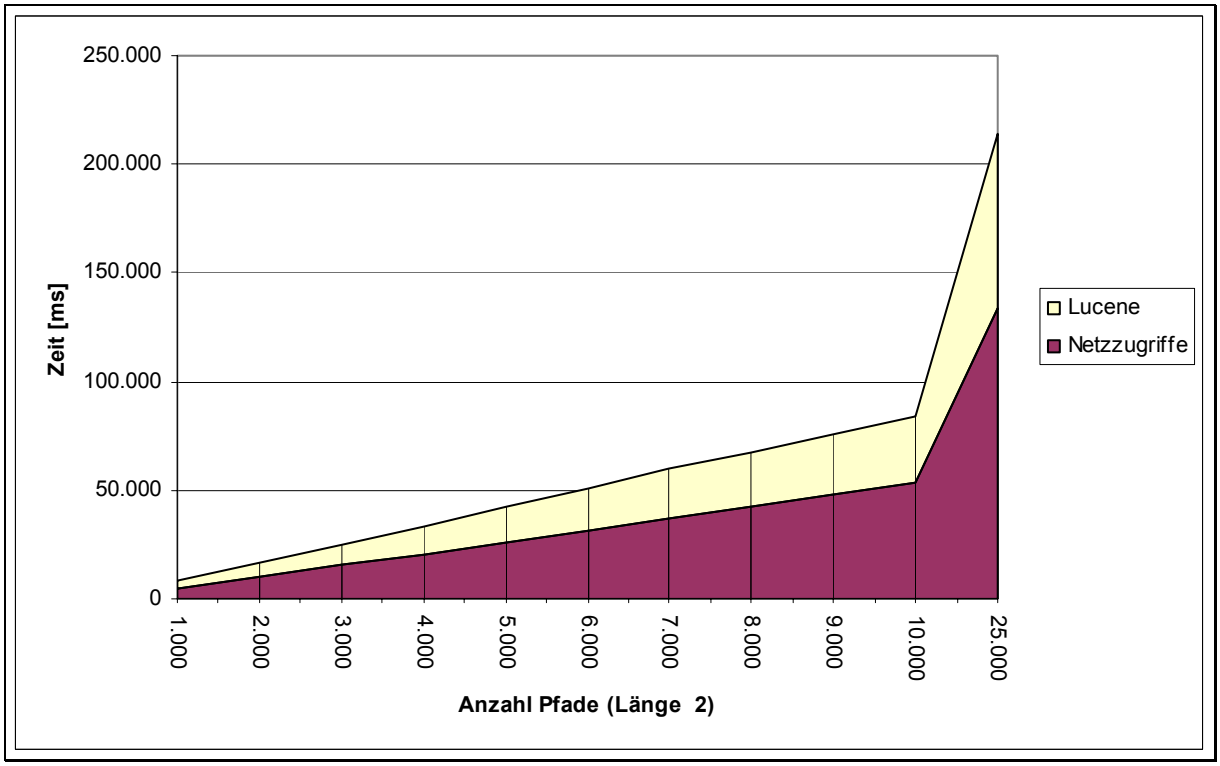


**Abbildung 6-5 Messwerte für CreatePathIndex (Pfadlänge 2)**

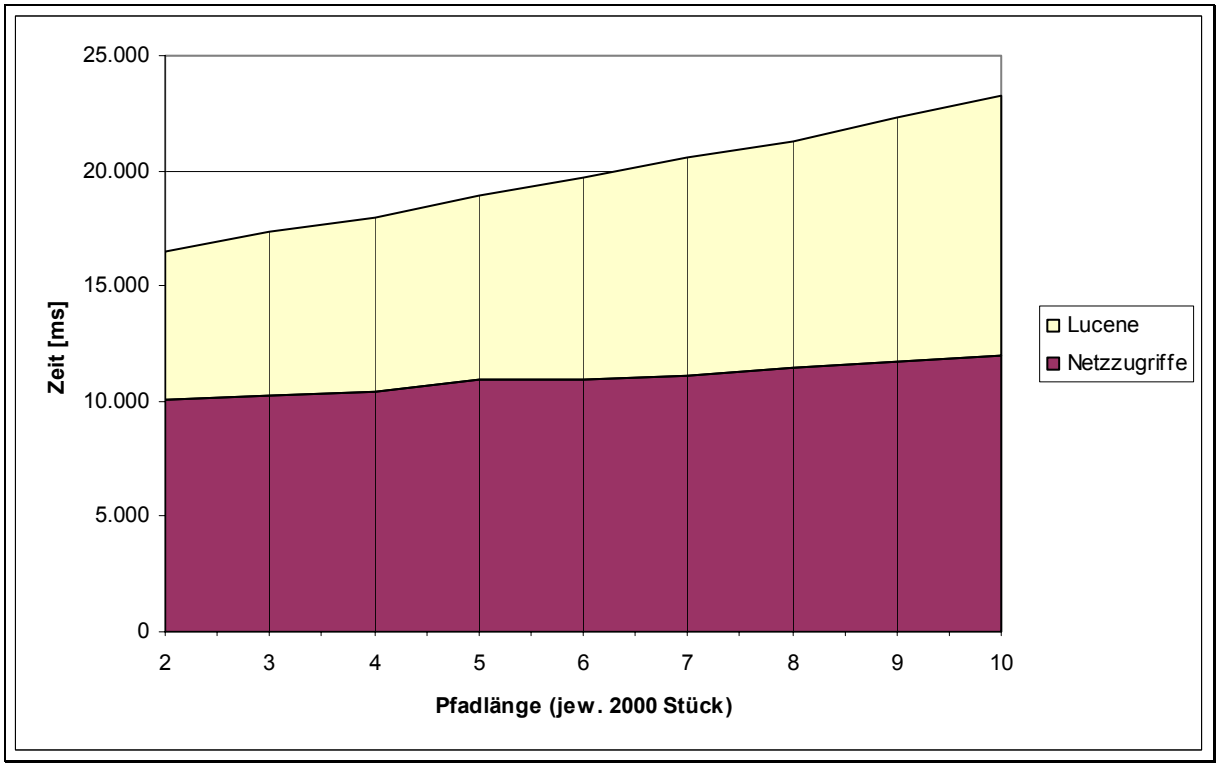


**Abbildung 6-6 Messwerte für CreatePathIndex (variierende Pfadlängen)**

Abbildung 6-5 zeigt zunächst die Messwerte für eine konstante Pfadlänge von 2 und verschiedenen Pfadanzahlen. Auch hier ist eine lineare Abhängigkeit zwischen Pfadanzahl und Aufwand zu beobachten. Allerdings liegt hier der Anteil der Netzzugriffe wesentlich höher (60,9% - 63,3%) als beim Anwendungsfall CreateNodeIndex. Dies ist darin begründet, dass beim Anwendungsfall CreateNodeIndex umfangreiche Daten (Knoteninhalte) indiziert werden müssen. Bei CreatePathIndex enthalten die Lucene-Dokumente nur OIDs wodurch der Aufwand für die Indizierung eines Dokumentes deutlich verringert wird.

In Abbildung 6-6 wird die Pfadlänge bei einer konstanten Pfadanzahl von 2000 Pfaden variiert. Das Diagramm zeigt tendenziell einen linearen Zusammenhang zwischen der Pfadlänge und dem Zeitaufwand. Der Anteil der Netz-Zugriffe verringert sich mit steigender Pfadlänge. Die Messungen der Pfadlänge 2 weisen einen Anteil von 60,9% auf, die Messungen der Pfadlänge 10 einen Anteil von 51,6%. Der Grund für diesen Trend ist ein steigender Aufwand für die Indizierung, da mit steigender Pfadlänge die Anzahl der Felder in den Lucene-Dokumenten, sowie die Länge des Feldes "edges" (vgl. Abschnitt 5.1.4) steigt.

Abbildung 6-7 zeigt die Gesamtaufwände bei variablen Pfadlängen und Pfadanzahlen.

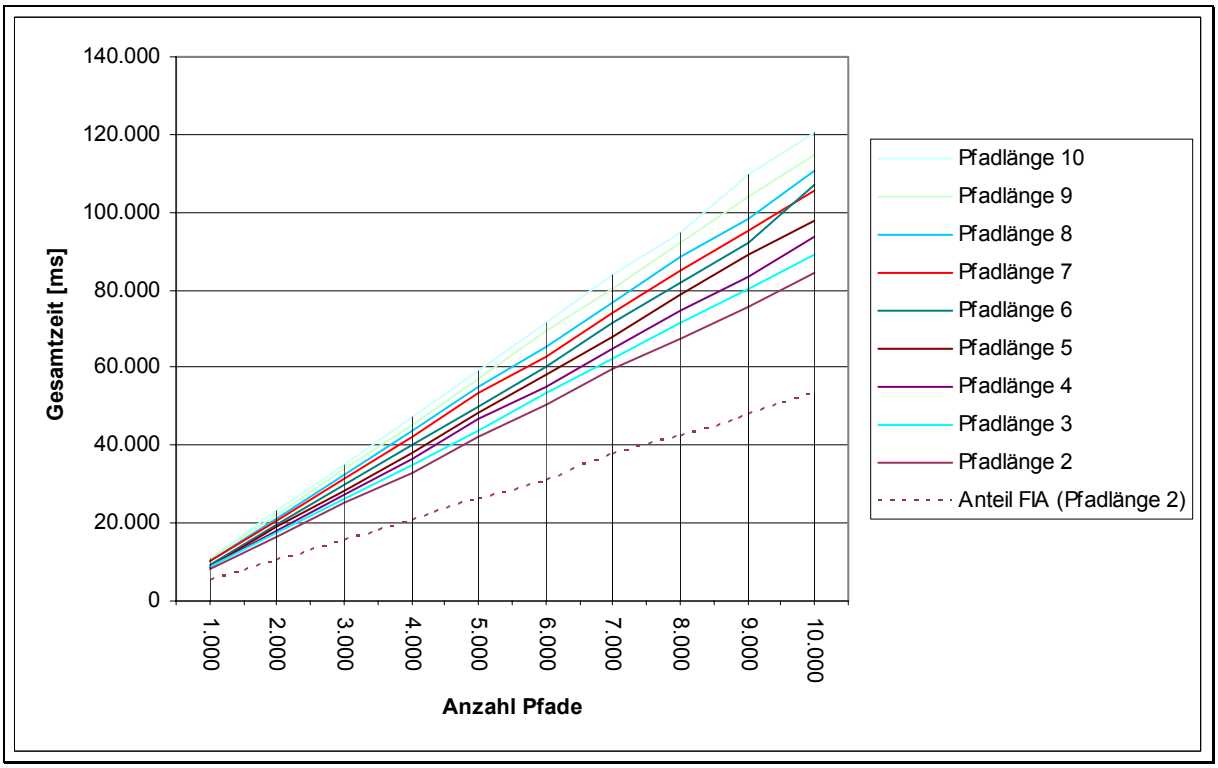


**Abbildung 6-7 Messwerte für CreatePathIndex (Gesamtwerte)**

Zur Orientierung ist der Anteil der Netzzugriffe für die Pfadlänge 2 gestrichelt eingetragen. Aus Darstellungsgründen (Skalierung der Y-Achse) sind im Diagramm die Messungen für 25.000 Pfade nicht enthalten. Der lineare Trend setzt sich jedoch weiter fort. Der Knick in der Linie der Pfadlänge 6 ist als Ausreißer anzusehen, da der Messwert bei 25.000 Pfaden diese Abweichung nicht aufweist.

### 6.5.2 Suche

Abbildung 6-8 zeigt die Messungen für OnlineSearch. Zu beobachten ist hier eine lineare Abhängigkeit, wobei größere Pfadlängen mehr Zeit benötigen als kürzere Pfadlängen. Die Messwerte für 500 Pfade sind in diesem Diagramm aus Darstellungsgründen (Skalierung der Y-Achse) nicht enthalten, der beobachtete lineare Trend setzt sich jedoch auch hier fort. Der abweichende Messwert für Pfadlänge 5 bei 100 Pfaden wird durch einen Ausreißer bei einer der drei Messungen verursacht.

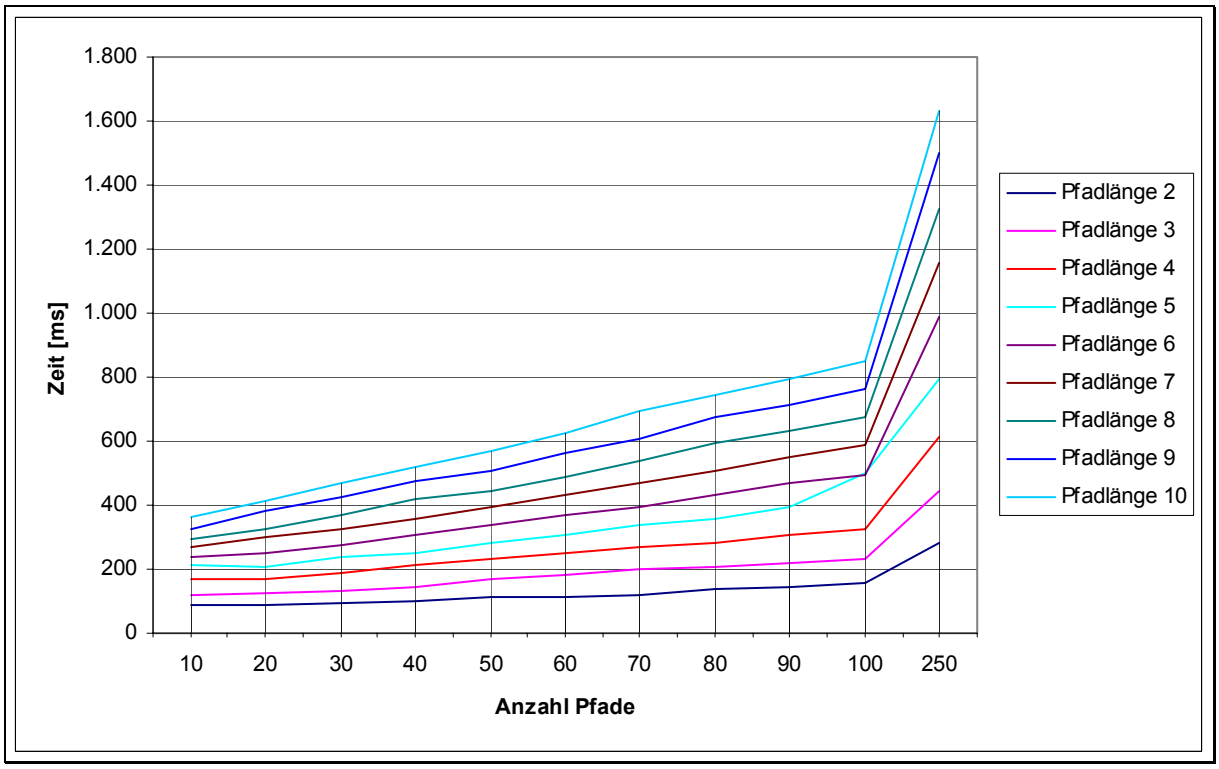


**Abbildung 6-8 Messwerte für OnlineSearch**

Abbildung 6-9 zeigt die Messergebnisse für dieselbe Suche unter Verwendung von Pfadindizes (IndexSearch). Der Einfluss der Indexverwendung ist bereits bei den ersten Messungen zu beobachten. So benötigt die indexgestützte Suche auf Pfaden der Länge 2 mit 10 Ergebnissen bereits ca. 40% weniger Zeit als die entsprechende Online-Suche. Der Geschwindigkeitsvorteil steigt extrem mit der Anzahl der Ergebnisse. Die indexgestützte Suche benötigt bei Pfadlänge 10 und 500 Ergebnis-Knoten nur noch 10% der Zeit, welche die Online-Suche benötigt. Die Verwendung eines Index eignet sich demnach besonders bei großen Ergebnismengen. Theoretisch müssten auch in diesem Diagramm lineare Steigerungen beobachtet werden, welche jedoch nur als Trend vorhanden sind. Der Grund für die Abweichungen liegt vermutlich in den geringen Absolutzeiten, da hier die Hintergrundaktivitäten der Java-VM (z.B. Speicherverwaltung) und des Betriebssystems verstärkt in die Messwerte einfließen.

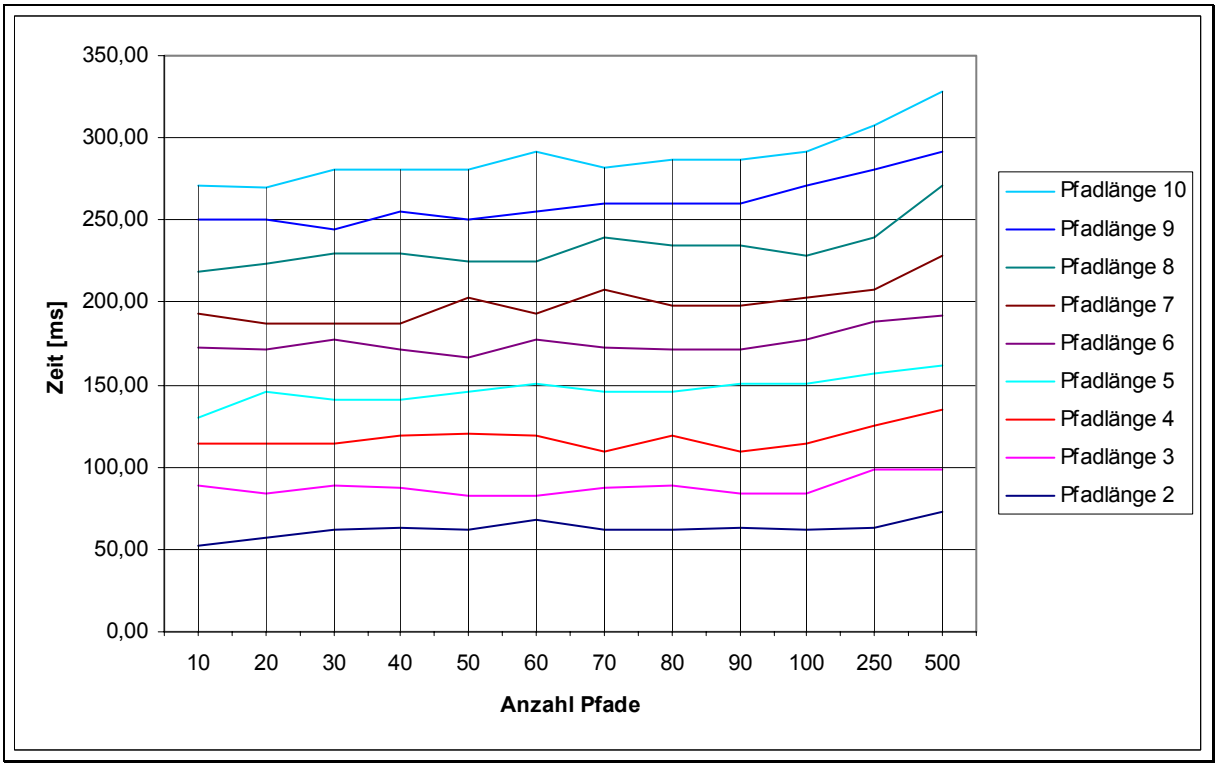


**Abbildung 6-9 Messwerte für IndexSearch**

### 6.5.3 Index-Pflege

Für die Index-Pflege werden die Aufwände für die explizite Index-Optimierung gesondert ausgewiesen, da die in Abschnitt 5.3.9 beschriebene Wahl des Zeitpunktes für die Optimierung genau diese Aufwände betrifft.

Die Messwerte für den Anwendungsfall *CreateNodes* sind in Abbildung 6-10 dargestellt. Der Aufwand für die Netzzugriffe ist fast nicht darstellbar. Da alle verwendeten Knoten direkt zuvor in die Datenbank eingefügt wurden, sind diese durch die Anwendung noch im Hauptspeicher vorhanden. Die Netzzugriffe enthalten demnach keinerlei Datenbankzugriffe. Der Aufwand ohne Optimierung zeigt einen linearen Trend. Die Aufwände für die Optimierung schwanken, da je nach Anzahl der neu eingefügten Knoten unterschiedliche Anzahlen von Segmenten kombiniert werden müssen. Bei 50 Knoten werden ein Segment mit 20.000 Dokumenten und fünf Segmente mit je 10 Dokumenten kombiniert. Bei 100 Knoten fällt der Aufwand, da die neuen Dokumente bereits durch die automatische Segment-Zusammenführung in ein einzelnes Segment zusammengefasst wurden. Dadurch muss dieses Segment nur noch mit dem vorhandenen Segment mit 20.000 Dokumenten kombiniert werden. Generell ist ein sublinearer Trend beim Aufwand für die Optimierung zu beobachten.

Abbildung 6-11 zeigt die durchschnittliche Aufwände pro Knoten auf. Der lineare Trend für den Aufwand pro Knoten ohne Optimierung ist hier schön zu sehen. Der Durchschnitt liegt bei 2,6 ms bei einer Spanne von 2,06 ms bis 3,63 ms. Der Trend weist eine leicht fallende Tendenz auf. Diese liegt darin begründet, dass konstante Aufwände, wie beispielsweise das Öffnen des Index für Schreibzugriffe, auf eine steigende Anzahl von Knoten verteilt werden. Die durchschnittlichen Aufwände mit Optimierung lassen einen potenziellen Zusammenhang vermuten. Dies ist nur dadurch zu erklären, dass der Aufwand für die Optimierung pro Segment mit steigender Segmentanzahl sinkt. Dies kann durch die folgenden Faktoren verursacht werden:

- Die Wahrscheinlichkeit identischer Terme wächst mit steigender Indexgröße, wodurch die Segmentgröße sublinear wächst (vgl. Abschnitt 2.8.1).

- Die Implementierung der Optimierung arbeitet stackbasiert. Zunächst werden alle vorhandenen Segmente auf einen Stack gelegt. Anschließend wird aus den obersten 10 Segmenten ein neues Segment erzeugt und wiederum auf den Stack gelegt. Dieses Segment wird anschließend mit den nächsten 9 Segmenten zusammengefasst. und das Ergebnis wieder auf den Stack gelegt. Dies wird fortgeführt bis alle Segmente zusammengefasst wurden. Da die jeweils neu erzeugten Segmente im Optimierungsprozess sofort wieder gelesen werden, können verstärkt die Caching-Mechanismen des Betriebssystems ausgenutzt werden, wodurch I/O-Zugriffe vermindert werden (vgl. [34]). Da die Größe der neuen Segmente kontinuierlich steigt, nimmt auch der Anteil der Daten des Index, welche aus dem Cache gelesen werden, stetig zu.

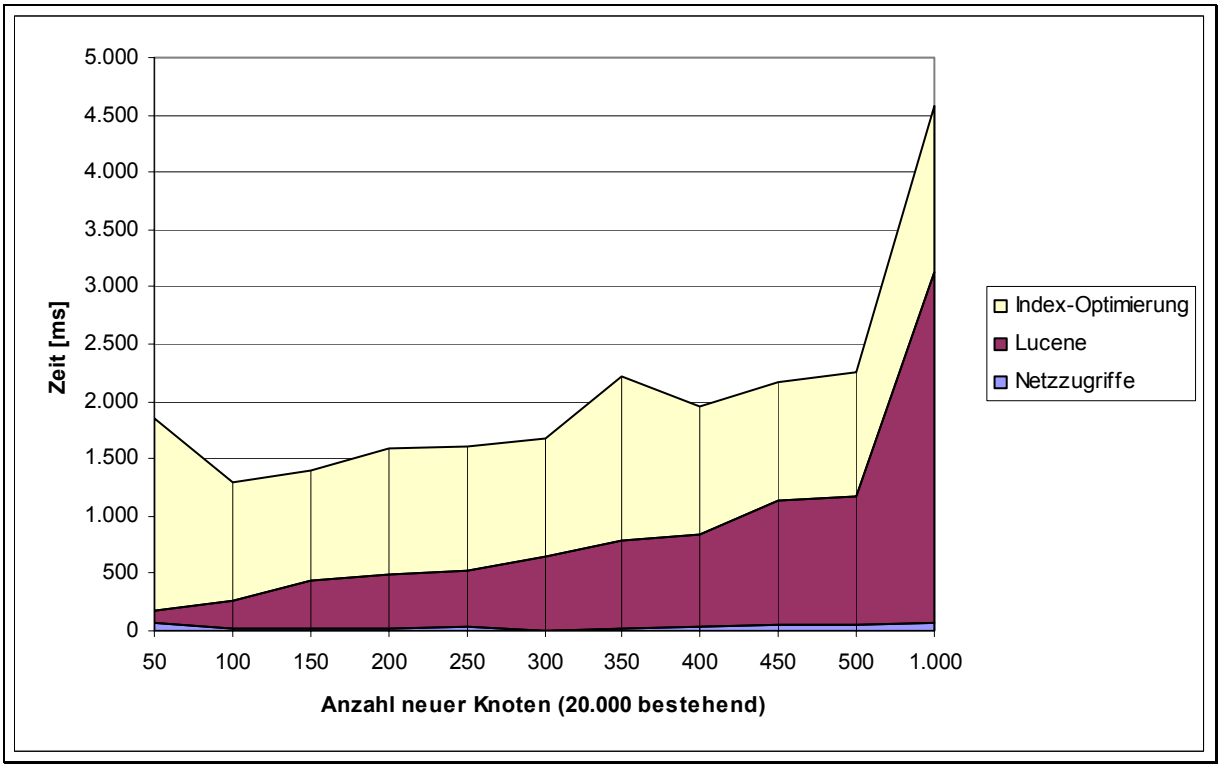


**Abbildung 6-10 Messwerte für CreateNodes (20.000 Knoten bestehend)**

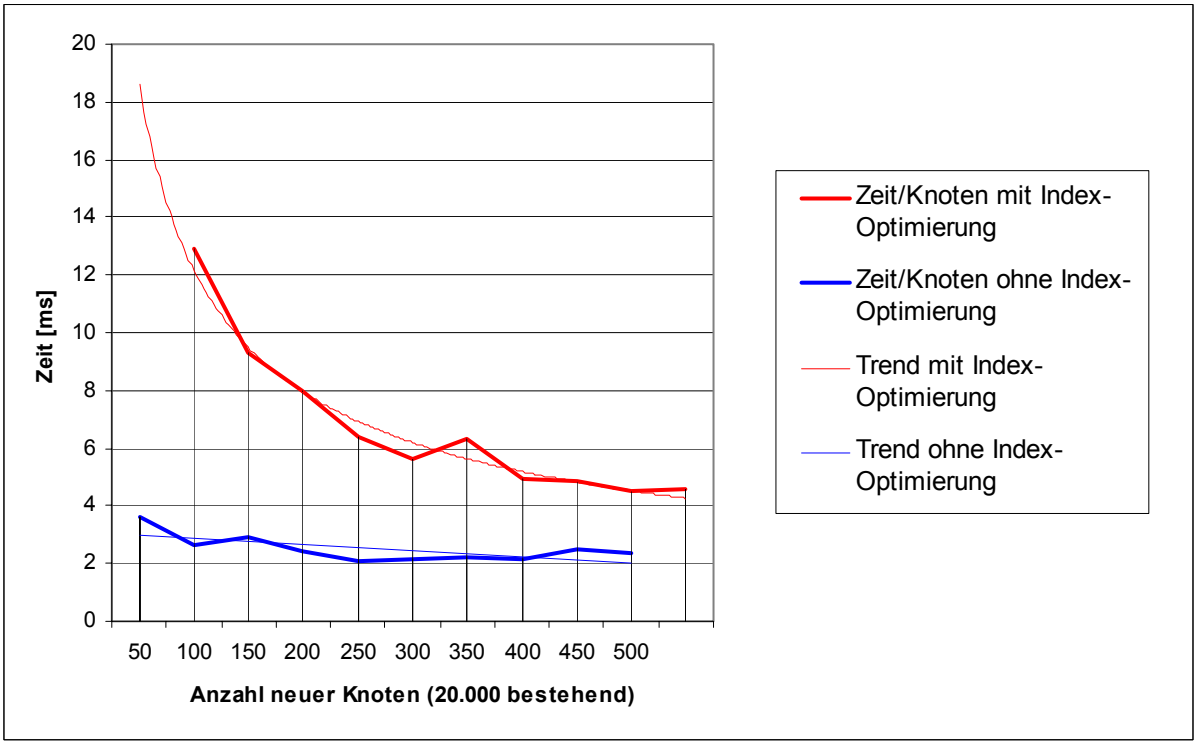


**Abbildung 6-11 Entwicklung der Zeit/Knoten bei CreateNodes (20.000 Knoten bestehend)**

Abbildung 6-12 zeigt die Werte aus der Vergleichsmessung basierend auf einem Modell mit 100.000 Knoten (CreateIndices-Modell mit 10.000 Pfaden). Der Vergleich zeigt, dass die Aufwände pro Knoten ohne Optimierung unabhängig von der bestehenden Netzgröße sind. Die Aufwände mit Optimierung weisen ebenfalls eine potenzielle Entwicklung auf, haben aber höhere Absolutwerte. Der Grund liegt in der fünffach größeren Datenmenge im Index, welche bei der Optimierung bewegt werden muss. Wie bereits

Abbildung 6-13 zeigt die Messwerte für den Anwendungsfall *CreatePaths* in Abhängigkeit von der Pfadanzahl. Im Diagramm ist eine feste Pfadlänge von 5 bei der Aktualisierung eines einzelnen Index dargestellt. Die Werte für die Pfadlänge 2 weisen ebenfalls einen linearen Trend auf. Bei 100 Pfaden ist der Einfluss der automatischen Segmentzusammenführung durch einen höhere Lucene- und geringeren Optimierungsaufwand sichtbar. Gegenüber CreateNodes ist der Aufwand für Optimierung hier ausgeglichen und wesentlich geringer. Dis ist darin begründet, dass die Dokumentgröße hier wesentlich geringer ist und gegenüber CreateNodes konstant bleibt.

Abbildung 6-14 zeigt die Entwicklung der Zeiten in Abhängigkeit von der Anzahl der betroffenen Indizes für verschiedene Pfadanzahlen. Erwartungsgemäß ist ein linearer Zusammenhang zwischen der benötigten Zeit und der Anzahl der betroffenen Indizes zu beobachten. Die benötigte Zeit wächst mit steigender Pfadanzahl geringer, aufgrund der sinkenden Optimierungszeit pro Pfad.

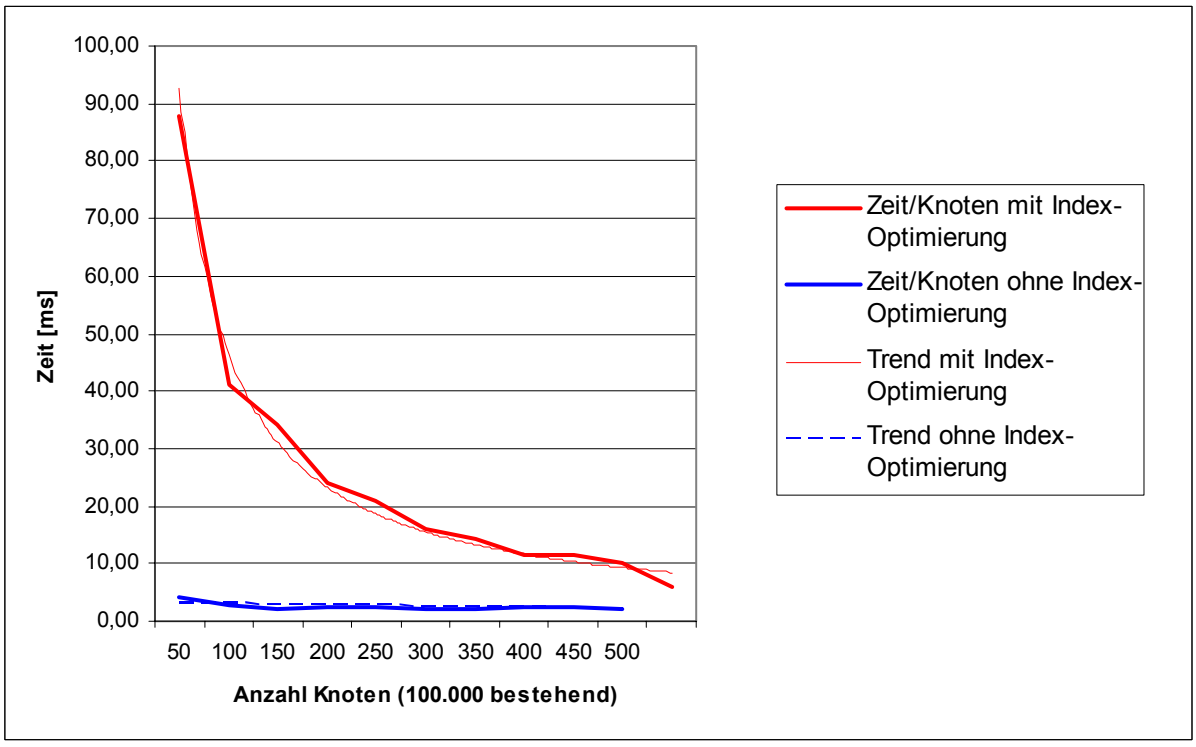


**Abbildung 6-12 Entwicklung der Zeit/Knoten bei CreateNodes (100.000 Knoten bestehend)**

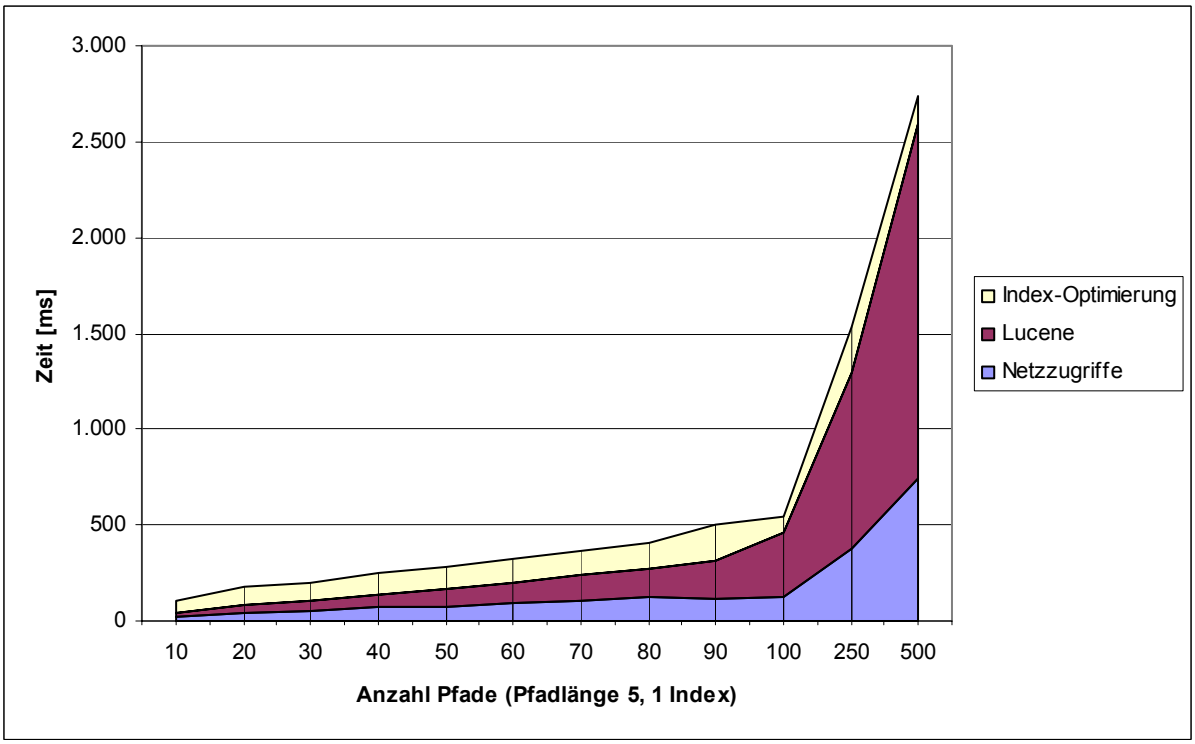


**Abbildung 6-13 Messwerte für CreatePaths (variable Pfadanzahl)**

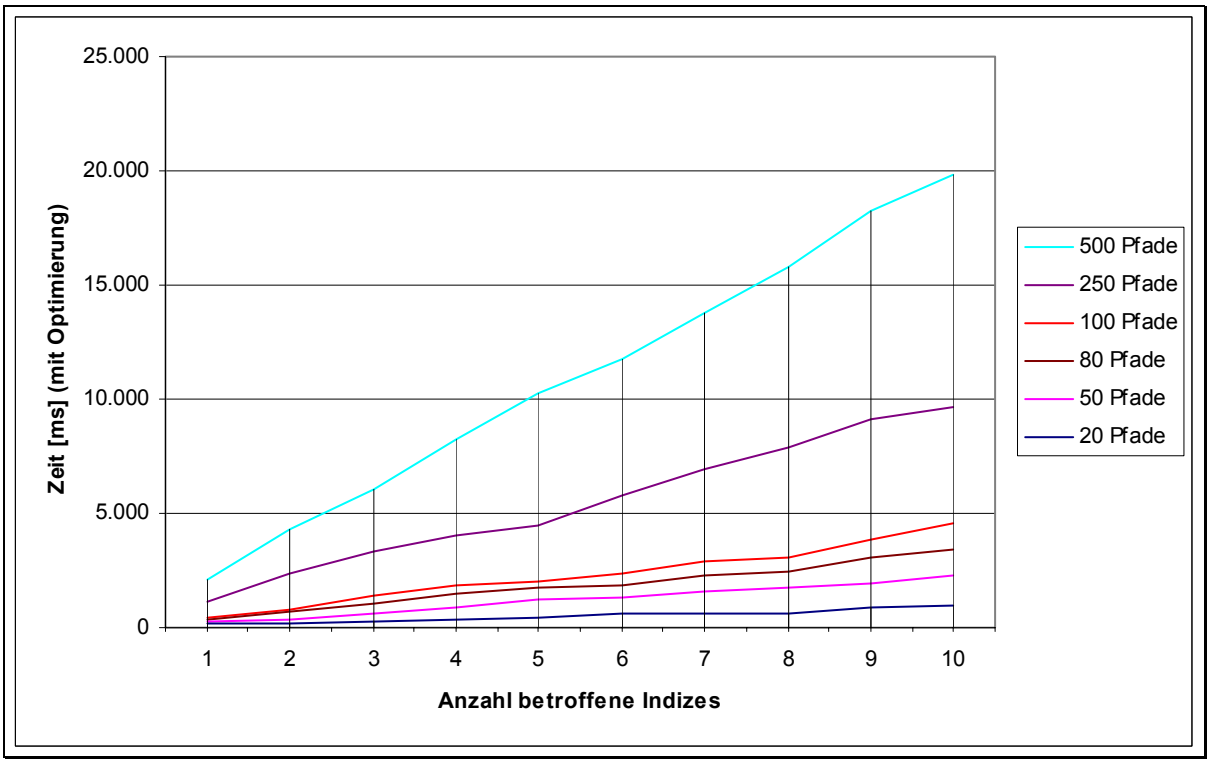


**Abbildung 6-14 CreatePaths: Gesamtzeit abhängig von Pfad- und Indexanzahl**

Um die Qualität der Messwerte zu bewerten wurden auf dem bestehenden realen Modell ebenfalls einige Anwendungsfälle ausgeführt und gemessen. Nachfolgend werden die dabei ermittelten Werte mit theoretischen Werten, welche durch Berechnung aus den vorhandenen Messwerten stammen verglichen.

Die Erstellung des Knotenindex (15.102 Knoten) benötigte 55 Sekunden ohne referenzierte externe Dateien und 61 Sekunden mit referenzierten Dateien. Der durch CreateNodeIndex ermittelte theoretische Wert ohne referenzierte Dateien liegt bei 52,6 Sekunden. Dies bedeutet eine Abweichung von ca. 4,6%, welche sich auf die Netzzugriffe konzentriert. Die Ursache liegt vermutlich in der aufwändigen Prüfung auf referenzierte Dateien, welche in den Testmodellen nicht vorgenommen wird.

Die Erstellung von Pfadindizes der Längen 2, 3 und 5 (4.501, 3.789 und 3.688 Pfade) benötigte ca. 16% weniger (28, 24 und 26 Sekunden) als die theoretische Zeit. Auch hier ist die Abweichung auf die Netzzugriffe konzentriert. Dies liegt an Unterschieden in der Struktur der für die theoretischen Messungen verwendeten Modelle (***Messmodelle***), da diese mehr pfadunabhängige Knoten enthalten. Außerdem sind die Knoten der Messmodelle nur in wenige organisatorische Einheiten (Units) unterteilt. Da der Zugriff auf alle Knoten pro Unit durch einen einzelnen Datenbankabruf erfolgt, sind hier die Datenmengen bei den Messmodellen größer.

Zusätzlich zur Indexerstellung wurden Suchen durchgeführt, welche allerdings aufgrund der unterschiedlichen Netzgröße nicht mit den theoretischen Werten verglichen werden können. Eine Suche über einen Index der Pfadlänge 5 mit angegebenen Startknoten und Bedingungen für den Endknoten erbrachte 1 Ergebnis und benötigte 31 ms. Eine Suche nach allen vorhandenen Parametern (also ohne Startknoten) mit dem Wert "A34.0" ergab 8 Ergebnisse und benötigte 343 ms.

## 6.6 Bewertung

Die ermittelten Werte zeigen im Wesentlichen die erwarteten Ergebnisse. Die indexgestützte Suche ist effizienter als die Online-Suche, jedoch wird der Vorteil durch die Notwendigkeit der Index-Erstellung und -Pflege relativiert. Aus diesem Grund lohnt sich die Indexverwendung nur bei mehrfachen Suchen auf der entsprechenden Kantentypfolge. Der Geschwindigkeitsvorteil der Indexsuche wächst mit steigender Pfadlänge und Anzahl der *Pfadkandidaten*.

Die Indexsuche eignet sich somit besonders für

- oft wiederkehrende Suchanfragen (bzgl. der *Kantentypfolge*)
- Suchen auf großen *Kantentypfolgen*
- Suchen mit vielen *Pfadkandidaten*
- Suchen ohne Startknoten (hier ist sie notwendig)

Bei Suchen auf kleinen Pfadlängen und mit wenigen Pfadkandidaten fällt der Zeitaufwand für eine Online-Suche nur unwesentlich ins Gewicht. Die Erstellung eines Pfadindex sollte daher dem Anwender überlassen werden, da die Indexpflege die Arbeitsgeschwindigkeit verringert

und nur der Anwender entscheiden kann, wie oft er Suchen auf bestimmten Kantentypfolgen benötigt.

Das größte Problem bei der Indexpflege ist der Zeitaufwand für die Optimierung der Indizes. Hier muss unbedingt ein Konzept entwickelt werden um die Arbeitsgeschwindigkeit nicht extrem zu bremsen. Eine Migration auf die kommende Lucene-Version 1.3 könnte hier aufgrund der Umstrukturierung des Indexformates eventuell den benötigten Zeitaufwand etwas verringern. Mit der Version 1.2 sind die in Abschnitt 5.3.9 genannten Varianten aufgrund des beschriebenen Bugs nicht einsetzbar. Eine viel versprechende Lösung für das Problem liegt in einer Änderung des Eventmechanismus durch die Zusammenfassung mehrerer Events zu einem Einzigen. Vorausgesetzt, dass Objekte welche angelegt und wieder gelöscht wurden nicht in diesem Event enthalten sind, ist bei diesem Verfahren auch der beschriebe Bug nicht relevant.

## Kapitel 7

# Zusammenfassung und Ausblick

In diesem Kapitel werden wichtige Ergebnisse dieser Arbeit kurz zusammengefasst und ein Ausblick auf die Implementierung in die bestehende Anwendung, sowie weitere Forschungsmöglichkeiten gegeben.

## *7.1 Zusammenfassung*

Der Maschinenbau ist, ebenso wie die Softwareentwicklung, einem immer stärker werdendem Wettbewerb ausgesetzt und muss daher die Entwicklungszeiten neuer Maschinen verringern bei gleichzeitiger Erhöhung der Qualität. Sowohl in der Softwaretechnik als auch im Maschinenbau wird versucht, dies durch eine Orientierung zu wiederverwendbaren Komponenten zu erreichen. Die *Föderale Informations Architektur* (FIA) konzentriert sich auf den Sondermaschinenbau und unterstützt durch eine Eclipse-basierte Anwendung eine baukastenbasierte Wiederverwendung. Zu diesem Zweck wird ein semantisches Netz aufgebaut, welches die wiederverwendbaren Komponenten zusammen mit Konfigurations-möglichkeiten enthält. Neue Maschinen werden in diesem Netz aus den bestehenden Komponenten des Baukastens modelliert. Nach der Modellierung werden die notwendigen Programme und Dokumentationen automatisch generiert. Die Komponenten werden in den vorhandenen spezialisierten Anwendungen (z.B. CAD- oder SPS-Systeme) erstellt. Die Daten werden in den entsprechenden Anwendungen belassen und durch die FIA lediglich referenziert.

Die FIA unterstützt nur rudimentäre Suchfunktionen. So ist eine Suche nur unter Kenntnis der Modellstruktur möglich und umfasst nur die in der FIA direkt abgelegten Daten. Referenzierte Daten (z.B. Steuerungsprogramme) können nicht in die Suche einbezogen werden. Diese Arbeit beschreibt ein Konzept zur Erweiterung der FIA um eine Volltextsuche, welche auch referenzierte Daten umfasst.

Suchfunktionen lassen sich in zwei grundlegende Kategorien einteilen. Bei der *Online-Suche* wird jedes Mal der gesamte Datenbestand in Bezug auf die definierten Suchbedingungen analysiert. Dieser Ansatz ist aufgrund der benötigten Zeit nur bei geringen Datenmengen (z.B. eine einzelne Datei) sinnvoll. Die *indexgestützte Suche* analysiert den Datenbestand nur einmalig und erzeugt eine auf Suchen zugeschnittene Datenstruktur (Index). Die Bearbeitung von Suchanfragen greift nur auf den Index zu, weswegen der Datenbestand nicht verfügbar sein muss. Dieses Verfahren wird eingesetzt für große Datenmengen (z.B. Internet-Suchmaschinen) oder Daten, welche nicht ständig verfügbar sind (z.B. Band-Archiv). Da die Analyse des Datenbestandes nur bei Änderungen erfolgt, können hier auch aufwändige Aufbereitungen vorgenommen werden. Für den Aufbau des Index existieren verschiedene Varianten, ebenso verschiedene Optionen, welche die Mächtigkeit einer Suche entscheidend beeinflussen und auch Suchen auf fehlerbehafteten Daten ermöglichen.

Mit dem ausführlich beschriebenen OpenSource-Framework *Lucene* der *Apache Software Foundation* können auf einfache Weise indexgestützte Suchfunktionen in eigene

Anwendungen implementiert werden. Das Framework bildet viele der in Internet-Suchmaschinen üblichen Funktionen ab und bietet eine effiziente Indizierung und Suche auch bei großen Datenmengen. Allerdings muss der Index durch ein relativ aufwändiges Verfahren optimiert werden. Dadurch eignet sich Lucene insbesondere für Suchen auf Daten welche sich nur selten ändern, so dass diese z.B. im täglichen Batchbetrieb indiziert werden können.

Für die Umsetzung der Volltextsuche in der FIA wurden zunächst die Anforderungen ermittelt und dokumentiert. Anschließend wurden drei verschiedene Varianten und intuitiv Kriterien für deren Vergleich und Vorbewertung entwickelt. Aufgrund dieser Vorbewertung wurden zwei Varianten ausgewählt und implementiert. Für eine empirische Bewertung wurde ein Modellgenerator entwickelt und durch diesen Netze verschiedener Größe erzeugt. Auf diesen Netzen wurden verschiedene Anwendungsfälle ausgeführt und deren Laufzeiten gemessen. Zusammenhänge und Abhängigkeiten von der Netzgröße wurden durch Diagramme dargestellt. Die ermittelten Werte belegen die Effizienz des eingesetzten Lucene-Frameworks sowohl bei der Indizierung als auch der Suche, zeigen aber auch die Probleme der Index-Optimierung auf. Die funktionalen Anforderungen an die Suchfunktion in der FIA konnten vollständig umgesetzt und auch durch weitere Möglichkeiten erweitert werden.

## 7.2 Integration in die bestehende Anwendung

Das erstellte Konzept kann prinzipiell zusammen mit der für die Messungen erstellten Implementierung mit geringem Aufwand in die Anwendung integriert werden. Dieser Abschnitt beschreibt die noch notwendigen Anpassungen und Erweiterungen.

Die momentan statisch codierten Konfigurationsparameter (z.B. Ablageort des Index) sollten durch eine dynamische Implementierung ersetzt werden, um hier Konflikte mit den existierenden Dateisystem-Strukturen der Anwender-PCs zu vermeiden. Beim Anwendungsstart muss die Schnittstelle zur Anwendung (MindListener) initialisiert werden, damit alle Änderungen am Netz sich in den Indizes widerspiegeln. Um dem Problem der Index-Optimierung zu begegnen, sollte das Konzept des MindListeners gemäß den Ausführungen in Abschnitt 6.6 angepasst werden. Sobald die kommende Lucene-Version 1.3 verfügbar ist, sollte eine Migration vorgenommen und basierend auf den Ausführungen in Abschnitt 5.3.9 der Optimierungszeitpunkt definiert werden. Für eine benutzerfreundliche Suche sind noch entsprechende Oberflächen zu erstellen.

## 7.3 Ausblick

An diese Arbeit anschließend wäre eine Untersuchung sinnvoll, welche die Auswirkungen einer Änderung der in Abschnitt 3.5.3 beschriebenen (in dieser Arbeit mit Standardwerten belegten) Parametern IndexWriter.`mergeFactor` und IndexWriter.`maxMergeDocs` auf die Effizienz der Indizierung ermittelt. Durch einen höheren Wert bei `mergeFactor` sind Verbesserungen der Laufzeit sowohl der Indizierungs als auch der Optimierung zu erwarten, da weniger Festplatten-Zugriffe notwendig sind. Dies verursacht aber eine Erhöhung des Hauptspeicherbedarfs, weswegen hier ein Kompromiss gefunden werden muss.

In Abschnitt 5.3.3 wurde eine alternative Möglichkeit der Erstellung eines Pfadindex beschrieben, welche geringere Laufzeiten als die bestehende Implementierung verspricht. Die genauen Auswirkungen können durch eine entsprechende Implementierung, Untersuchung und Vergleich mit der bestehenden Implementierung ermittelt werden. Durch eine ähnliche

Implementierung in der Online-Suche kann die Einschränkung auf Suchen mit angegebenen Startknoten eventuell aufgehoben werden (vgl. Abschnitte 5.1.4 und 5.1.5).

## Anhang A

# Glossar und Abkürzungsverzeichnis

**AWL**

Abkürzung für ***A**n**w**eisungs**l**iste*. Es handelt sich dabei um eine Programmiersprache für *SPS*, welche in der DIN 19239 definiert ist. Sie ist eine der internen Maschinensprache sehr nahe kommende Programmiersprache und bietet, neben Grundfunktionen zur Lösung von Steuerungsaufgaben, einen sehr umfangreichen erweiterten Befehlssatz. ([15])

**Batchbetrieb**

Der *Batchbetrieb* dient der asynchronen Ausführung zeitintensiver Aufgaben. Hierzu werden während der Arbeit mit dem Anwender für die durchzuführenden Aufgaben Aufträge (*Batches*) erstellt, welche zu einem späteren Zeitpunkt (z.B. in der Nacht) ausgeführt werden. Dieses Verfahren wird beispielsweise auf Großrechnern von Banken eingesetzt, bei welchen Buchungen am Tag nur erfasst und in der Nacht durchgeführt werden.

**BLOB**

Abkürzung für *Binary Large Object,* bezeichnet einen Datentyp welcher in Datenbanken zur Darstellung großer binärer Daten, wie beispielsweise Dateien verwendet wird.

**CAD**

Abkürzung für *Computer-Aided-Design*. Bezeichnet den EDV-gestützten Entwurf mechanischer Bauteile.

**FIA**

Abkürzung für *Föderale Informations Architektur*, die bestehende Anwendung

**FQN**

Abkürzung für *Fully Qualified Name*. Der FQN identifiziert eine Java-Klasse eindeutig, indem der Packagename und der Klassenname in einem String zusammengefasst werden. Das Format für einen FQN ist *Package.Klasse* (z.B. *java.lang.String*)

**IETF**

Abkürzung für Internet Engineering Task Force.

Eine offene internationale Gemeinschaft, welche sich um die Architektur und die reibungslose Funktionsfähigkeit des Internets kümmert.

**HTML**

Abkürzung für ***H**yper**t**ext **M**arkup **L**anguage*.

**OID**

Abkürzung für ***O**bject **Id**entifier*. Die *OID* identifiziert in der *FIA* eindeutig ein Objekt (z.B. Knoten, Kante oder Kantentyp) und wird in der Datenbank als Primärschlüssel verwendet.

**PDF**

Abkürzung für ***P**ortable **D**ocument **F**ormat*. PDF ist ein Dokumentenformat, welches auf unterschiedlichen Plattformen angezeigt werden kann. Durch die Möglichkeit die Art der erlaubten Zugriffe zu definieren (z.B. nur lesen, lesen und ändern, Inhalt entnehmen, ...) hat dieses Format eine weite Verbreitung gefunden.

**RTF**

Abkürzung für ***R**ich **T**ext **F**ormat*. Dabei handelt es sich um ein textbasiertes Datenformat für formatierte Dokumente

**SPS**

Abkürzung für ***S**peicher-**P**rogrammierbare-Steuerung*. Eine Einführung ist in [15] gegeben.

**Stack**

Ein Datencontainer, auch Stapel oder Keller genannt, welcher nach dem Last-In-First-Out (LIFO)-Prinzip arbeitet. Die Sortierung der Elemente wird beibehalten, die einzigen elementaren Operationen sind push (neues Element oben auf den Stack legen) und Pop (oberstes Element vom Stack zurückgeben).

**Technische Schnittstelle**

Eine technische Schnittstelle dient der Anbindung anderer Komponenten oder Systeme. Im Gegensatz zu einer nicht-technischen Schnittstelle (wie beispielsweise einer Benutzerschnittstelle) sind keine direkten Interaktionen mit dem Anwender oder Peripheriegeräten (wie Drucker oder Bildschirm) vorhanden.

**TREC**

Abkürzung für *Text Retrieval Conference*. Es handelt sich hierbei um eine sehr große Dokument-Sammlung welche weltweit zum experimentellen Vergleich im Bereich des Information Retrievals verwendet wird. Die Trec wird durch eine Reihe verschiedener Magazine und Zeitschriften, wie beispielsweise dem *Wall Street Journal*, aufgebaut

**UInt64**

Bezeichnung für eine 64 Bit lange vorzeichenlose Integer-Zahl. Diese kann somit einen Zahlenbereich von 0 bis ca. $1{,}8*10^{19}$ darstellen.

**UML**

Abkürzung für *Unified Modelling Language*, eine standardisierte Notation für graphische Darstellungen bei objektorientierten Sprachen. Eine Beschreibung ist in [43] zu finden.

**WebCrawler**

Ein WebCrawler durchsucht selbständig ganze Internet-Sites und fügt die enthaltenen Dokumente in den Index einer Suchmaschine ein. Die Ermittlung der Dokumente kann durch Verfolgung von Links erfolgen oder durch Analyse der Dateisysteme.

**XML**

Abkürzung für ***E****xtended* ***M****etadata* ***L****anguage*.

## Anhang B

# Literaturverzeichnis


[1] APACHE.ORG:
*Homepage des Lucene-Projektes.*
http://jakarta.apache.org/lucene (3.4.2003)

[2] APACHE.ORG:
*JavaDoc der Lucene-Klassen.*
http://jakarta.apache.org/lucene/docs/api/index.html (9.4.2003)

[3] APACHE.ORG:
*Lucene-FAQ, Searching Section.*
http://lucene.sourceforge.net/cgi-bin/faq/faqmanager.cgi?toc=faq (21.4.2003)

[4] APACHE.ORG:
*Lucene Index-Dateiformate.*
http://jakarta.apache.org/lucene/docs/fileformats.html (4.4.2003)

[5] APACHE.ORG:
*Syntax der Lucene-Suche.*
http://jakarta.apache.org/lucene/docs/queryparsersyntax.html (18.4.2003)

[6] BAEZA-YATES, RICARDO; NAVARRO GONZALO:
*Block Addressing Indices for Aproximate Text Retrieval.*
Conference on Information and Knowledge Management. Las Vegas, 1997
http://doi.acm.org/10.1145/266714.266719 (12.5.2003)

[7] BAEZA-YATES, RICARDO; RIBEIRO-NETO, BERTHIER:
*Modern Information Retrieval.*
Addison Wesley, 1999, ISBN 0-201-39829-X

[8] CAUMANNS, JÖRG:
*A Fast and Simple Stemming Algorithm for German Words.*
Center für Digitale Systeme, Freie Universität Berlin, 1999, Report B99-16
http://www.inf.fu-berlin.de/inst/pubs/tr-b-99-16.abstract.html (11.4.2003)

[9] DETEMEDIEN, DAS TELEFONBUCH:
*Online-Telefonauskunft.*
http://www.telefonbuch.de (18.7.2003)

[10] ECLIPSE.ORG:
*Homepage des Projekts.*
http://www.eclipse.org (24.4.2003)

[11] ENSELING, OLIVER:
*Build your own languages with JavaCC.*
JavaWorld.com, 2000
http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html (13.4.2003)

[12] FOEDERAL.ORG:
*Homepage des Projekts.*
http://www.foederal.org (24.4.2003)

[13] FOEDERAL.ORG:
*Beschreibung der Problematik im Maschinen- und Anlagenbau.*
http://www.foederal.org/0problem.htm (24.4.2003)

[14] FRIEDL, JEFFREY E. F.:
*Reguläre Ausdrücke.*
O'Reilly, 2001, ISBN 3-930673-62-2

[15] FRIELER, DIETER:
*Einführung SPS.*
http://www.studet.fh-muenster.de/~diefrie/einfh.html (13.7.2003)

[16] GAMMA, ERICH; HELM, RICHARD; JOHNSON, RALPH; VLISSIDES, JOHN:
*Design Patterns - Elements of Reusable Object-Oriented Software, Deutsche Ausgabe.*
Addison-Wesley Longman Inc., 1997, ISBN 0-201-63361-2

[17] GOETZ, BRIAN:
*The Lucene search engine: Powerful, flexible and free.*
JavaWorld.com, 2000
http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene_p.html, (3.4.2003)

[18] GOOGLE DEUTSCHLAND GMBH:
*Google-Suchmaschine.*
www.google.de (13.4.2003)

[19] GOSPODNETIC, OTIS:
*Advanced Text Indexing with Lucene.*
OnJava, 2003
http://www.onjava.com/pub/a/onjava/2003/03/05/lucene.html (3.4.2003)

[20] GOSPODNETIC, OTIS:
*Introduction to Text Indexing with Apache Jakarta Lucene.*
OnJava, 2003
http://www.onjava.com/pub/a/onjava/2003/01/15/lucene.html (3.4.2003)

[21] HÄRDER, THEO; RAHM, ERHARD:
*Datenbanksysteme – Konzepte und Techniken der Implementierung.*
Springer-Verlag, 1999, ISBN 3-540-65040-7

[22] HARDT, MANFRED:
*Wo war denn noch gleich ...?.*
JavaMagazin September 2002, 39-46
Software und Support Verlag GmbH

[23] HEAPS, H.S:
*Information Retrieval - Computational and Theoretical Aspects.*
Academic Press, 1978, ISBN 0-12-335750-0

[24] HIRSCHBERG, DAN:
*Serial Computations of Levenshtein Distances.*
In: Pattern Matching Algorithms, p. 123-141. APOSTOLICO, A.; GALIL, Z.
Oxford University Press 1997
http://citeseer.nj.nec.com/hirschberg97serial.html (20.8.2003)

[25] INTERNET ENGINEERING TASK FORCE (IETF):
*Multipurpose Internet Mail Extensions.*
Request for Comments (RFC) Nummer 2045, 2046 und 2047
http://www.ietf.org/rfc/rfc2045.txt (13.7.2003)
http://www.ietf.org/rfc/rfc2046.txt (13.7.2003)
http://www.ietf.org/rfc/rfc2047.txt (13.7.2003)

[26] INTERNET ENGINEERING TASK FORCE (IETF):
*UTF-8, a transformation format of ISO 10646.*
Request for Comments (RFC) Nummer 2279
http://www.ietf.org/rfc/rfc2279.txt (13.7.2003)

[27] JAVA.NET:
*Java Compiler Compiler (JavaCC) - The Java Parser Generator, Projekt-Homepage.*
https://javacc.dev.java.net/ (3.8.2003)

[28] MANBER, UDI; WU, SUN:
*GLIMPSE: A Tool to Search Through Entire File Systems.*
Proceedings of the USENIX Winter 1994 Technical Conference, p. 23-32
San Francisco, Januar 1994
http://citeseer.nj.nec.com/manber94glimpse.html (30.7.2003)

[29] NAVARRO, GONZALO; BAEZA-YATES, RICARDO; SUTINEN, ERKKI; TARHIO, JORMA:
*Indexing Methods for Approximate String Matching.*
IEEE Data Engineering Bulletin, Vol. 24.4, p. 19 - 27 (2001)
http://citeseer.nj.nec.com/navarro00indexing.html (31.7.2003)

[30] NEC RESEARCH INSTITUTE:
*Scientific Literature Digital Library (CiteSeer).*
http://www.researchindex.org (30.7.2003)

[31] PAGE, LAWRENCE; BRIN, SERGEY; MOTWANI, RAJEEV; WINOGRAD, TERRY:
*The Pagerank Citation Ranking: Bringing Order to the Web.*
Stanford Digital Library Technologies Project, 1998
http://citeseer.nj.nec.com/page98pagerank.html (30.7.2003)

[32] POET SOFTWARE GMBH:
*FastObjects by Poet.*
http://www.fastobjects.com (15.7.2003)

[33] PORTER, MARTIN:
*An algorithm for suffix stripping.*
In: Readings in Information Retrieval, p. 313-316
Morgan Kaufmanns Publishers, ISBN 1-55860-454-5

[34] RUSSINOVICH, MARK:
*Inside the Cache Manager.*
Windows NT Magazine, Oktober 1998
Penton Media, Inc.
http://www.winntmag.com/Articles/Print.cfm?ArticleID=3864 (22.8.2003)

[35] SALTON, GERARD; FOX, EDWARD A.; WU, HARRY:
*Extended Boolean Information Retrieval.*
Communications of the ACM Volume 26, Issue 11, p. 1022-1036, 1986
ISSN: 0001-0782
http://doi.acm.org/10.1145/182.358466 (27.7.2003)

[36] SCHULZ, KLAUS; MIHOV, STOYAN:
*Fast String Correction with Levenshtein Automata.*
International Journal on Document Analysis and Recognition Vol. 5 Nr. 1, Nov 2002, pp. 67-85
http://citeseer.nj.nec.com/417624.html (20.8.2003)

[37] SEARCHTOOLS.COM:
*Search Tools Code Library Report - Lucene.*
http://www.searchtools.com/tools/lucene.html (10.4.2003)

[38] SEDGEWICK, ROBERT:
*Algorithmen.*
Addison-Wesley, 1991, ISBN 3-89319-301-4

[39] SHECTER, ROBB:
*Design by Interface.*
In: Dr. Dobb's Journal, 24 (1999) 2, pp. 96-101
http://www.ddj.com/documents/s=906/ddj9902j/9902j.htm (21.4.2003)

[40] SOMMERGUT, WOLFGANG:
*Bei Google laufen alle Fäden zusammen.*
Computerwoche Nr. 16 vom 18. April 2003, p. 10f
http://www.computerwoche.de/index.cfm?pageid=267&type=ArtikelDetail&id=80111819 (30.7.2003)

[41] STADLER, BJÖRN:
*Anfragesprache für ein semantisches Netz.*
Diplomarbeit Nr. 1963, Institut für verteilte und parallele Höchstleistungsrechner, Universität Stuttgart, 2002
ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-1963/DIP-1963.pdf (25.4.2003)

[42] SUN MICROSYSTEMS:
*Code Conventions for the Java Programming Language.*
http://java.sun.com/docs/codeconv/ (21.4.2003)

[43] TOGETHERSOFT INC.:
*Practical UML - A Hands-On Introduction for Developers.*
http://www.togethersoft.com/services/practical_guides/umlonlinecourse/index.html (2.8.2003)

[44] UNIVERSITY OF WAIKATO:
*NZDL - The New Zealand Digital Library*
http://www.nzdl.org (17.7.2003)

[45] VERITY INC.:
*Verity Information Server*
www.verity.com (6.5.2003)

[46] WISSTEN, IAN H.; MOFFAT, ALISTAIR; BELL, TIMOTHY C.:
*Managing Gigabytes - Compressing and Indexing Documents and Images, Second Edition*
Morgan Kaufmann Publishers, 1999, ISBN 1-55860-570-3

[47] ZOBEL, JUSTIN; DART, PHILIP:
*Phonetic String Matching: Lessons from Information Retrieval.*
19th Inter. Conference on Research and Development in Information Retrieval (SIGIR'96), p. 166-172, Aug. 1996.
http://citeseer.nj.nec.com/zobel96phonetic.html (18.7.2003)

Ich versichere, dass ich diese Arbeit selbständig verfasst und
nur die angegebenen Hilfsmittel verwendet habe

_____

Michael Wenig