Diplomarbeit

Integration von heterogenen Informationssystemen am Beispiel der Java Content Repository Spezifikation und der CoreMedia Smart Content Technology

November 2004

Verfasser: Daniel Spilker
Gärtnerstraße 92b
20253 Hamburg
spilker@tu-harburg.de

Betreuung: Prof. Dr. Joachim W. Schmidt
Prof. Dr. Ralf Möller
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

TUHH
Technische Universität Hamburg-Harburg

Zusammenfassung

In dieser Arbeit wird versucht, eine Schnittstellendefinition für Repository-Systeme zu finden, die es ermöglicht, viele Dienste von Informationssystemen abzubilden. Es wird untersucht, welche Dienste unterstützt werden müssen, um eine möglichst große Zahl von Systemen abzudecken. Mit der gefundenen Schnittstelle wird eine Integration von Informationssystemen in die CoreMedia Smart Content Technology entworfen und durchgeführt. Das Ziel dieser Implementierung ist es, unterschiedliche Informationssysteme so in vorhandene Architektur einzubinden, dass alle Dienste der CoreMedia Smart Content Technology auch mit den Informationen aus den integrierten Systemen genutzt werden können.

Danksagung

Allen, die zum Gelingen dieser Arbeit beigetragen haben, möchte ich danken.

Insbesondere bedanke ich mich bei Herrn Prof. Dr. Joachim W. Schmidt für die Unterstützung, die Anregungen und die Betreuung dieser Arbeit. Ich danke Herrn Prof. Dr. Ralf Möller, der sich zur Übernahme des Zweitgutachtens bereiterklärt hat, und den Mitarbeitern des Arbeitsbereichs Softwaresysteme an der TU Hamburg-Harburg für die stete Hilfsbereitschaft.

Mein besonderer Dank gilt Herrn Sören Stamer und den Mitarbeitern der CoreMedia AG, die durch fachliche Diskussion und ein sehr angenehmes Arbeitsklima das Entstehen dieser Arbeit gefördert haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In Organisationen, wie Unternehmen, Universitäten oder Regierungsstellen entstehen in heutiger Zeit große Mengen von Informationen. Diese Informationen werden mit Mitteln der elektronischen Datenverarbeitung in Informationssystemen organisiert, verwaltet und gespeichert.

Informationssysteme sind eine spezielle Klasse von Software-Systemen, die sich durch die charakteristischen Eigenschaften Persistenz, Quantität, Reaktivität und Integrität definieren lassen. Ein Informationssystem verwaltet einen umfangreichen, strukturieren (teilweise auch semi-strukturierten) Informationsbestand (*Quantität*), dessen Lebensdauer nicht durch die Lebensdauer einzelner individueller Geschäftsprozesse beschränkt ist (*Langlebigkeit, Persistenz*). Der Informationsbestand wird durch das Informationssystem basierend auf eingehenden Informationen ständig aktuell gehalten. Ein Informationssystem kann auf Anfragen antworten und in der Umgebung, in die es eingebunden ist, Aktionen auslösen (*Reaktivität*). Die *Integrität* des Informationsbestandes sowie der ein- und ausgehenden Informationen wird durch das Informationssystem gewahrt.

Als Computer noch teure und raumfüllende Geräte waren, die nur große Unternehmen, wie Banken und Versicherungen bezahlen konnten, gab es zunächst nur ein einziges oder wenige zentrale Informationssysteme in jeder Organisation. Durch den schnellen Fall der Kosten und aufgrund der Miniaturisierung von Computern wurden Informationssysteme bald nicht nur auf Unternehmensebene eingesetzt, sondern auch auf Abteilungs- und Arbeitsgruppenebenen bis hinunter zu einzelnen Arbeitsplatzrechnern. Dadurch entstand eine Vielzahl von Informationssystemen für unterschiedliche Informationsbestände, die von verschiedenen Benutzern verwendet werden.

Damit ein Informationssystem mit den menschlichen Benutzern in seiner Umgebung interagieren kann, gibt es textuelle oder graphische Benutzeroberflächen, die es erlauben Information abzufragen, zu erzeugen, zu modifizieren oder zu löschen. Interaktion mit computerbasierten Systemen wird über Bibliotheksschnittstellen oder Kommunikationsdienste abgewickelt. In modernen Client/Server- oder 3-Schichten-Architekturen sind die Benutzeroberflächen nicht direkt ein Teil des Informationssystems, befinden sich nicht im Kern des Systems, sondern sind in einer Präsentationsschicht angeordnet, die mit dem Informationssystem kommuniziert. Das Informationssystem ist hierbei ein Diensterbringer (Server) und die Systeme in der Umgebung des Informationssystems, die mit diesem kommunizieren werden als Klienten bezeichnet.

Für jedes Informationssystem werden spezielle Klienten entworfen. Software-Entwickler benutzen integrierte Entwicklungsumgebungen, um auf Software Configuration Management Systeme zuzugreifen, Redakteure benutzen Text- und Grafikeditoren, um Inhalte in Content Management Systeme einzugeben, Verwaltungsangestellte benutzen die Klienten des *Enterprise Resource Planning* (ERP) Systems, in welches sie Daten eintragen. In der Regel kommuniziert jeder Klient mit einem Informationssystem über spezielle Kommunikationsdienste oder Bibliotheksschnittstellen, die für dieses Informationssystem entwickelt worden sind.

Um einen Mehrwert aus den in den verschiedenen Systemen enthaltenen Informationen zu generieren und die Informationen einem größeren Benutzerkreis zugänglich zu machen,

wurden in den letzten Jahren Portalsysteme (oder kurz Portale) [Weg02] entwickelt. Portale versuchen alle verfügbaren Informationen unter einer einheitlichen Benutzeroberfläche zusammenzufassen. Ein Portalsystem ist also Klient von mehreren Informationssystemen.

Da jedes Informationssystem eine eigene Kommunikationsschnittstelle mit sich bringt, muss für jedes Portalsystem oder generell jeden Klienten, der nicht speziell für die Kommunikation mit einem Informationssystem entwickelt wurde, ein Adapter zur Kommunikation mit dem Informationssystem entwickelt werden. Interagieren n Klienten K_i $(1 \leq i \leq n)$ mit m Serversystemen S_j $(1 \leq j \leq m)$, sind im schlimmsten Fall $n \times m$ Adapter notwendig, um die Kommunikation herzustellen. Abbildung 1 zeigt dieses Szenario. Die Kanten in dem Diagramm stellen die Adapter dar.

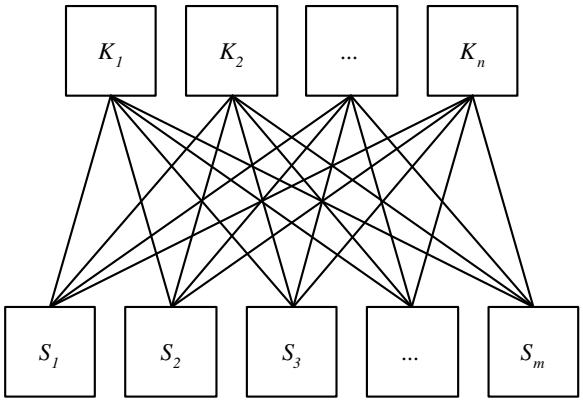

Abbildung 1: Integration von Informationssystemen ohne einheitliche Schnittstelle

Gäbe es nun eine einheitliche Schnittstelle I, die von den vielen verschiedenen Schnittstellen der unterschiedlichen Informationssysteme abstrahiert, wären nur $n + m$ Adapter notwendig (siehe Abbildung 2). Für die Interaktion der n Klienten mit der Schnittstelle werden n Adapter benötigt und m Adapter, um die einheitliche Schnittstelle auf die m Informationssysteme abzubilden.

Das Ziel dieser Arbeit ist es, eine solche Schnittstelle I zu finden. Diese Schnittstelle sollte möglichst viele Dienste von Informationssystemen abbilden. Welche Dienste dazuzählen wird hier untersucht. Mit der gefundenen Schnittstelle soll dann eine Implementierung der Integration von Informationssystemen in die CoreMedia Smart Content Technology entworfen und durchgeführt werden. Das Ziel dieser Implementierung ist es, unterschiedliche Informationssysteme so in vorhandene Architektur einzubinden, dass alle Dienste der CoreMedia Smart Content Technology auch mit den integrierten Systemen genutzt werden können.

Das zweite Kapitel zeigt die verschiedenen Klassen von Informationssystemen im Überblick. Es wird auf einige ausgewählte Dienste und Merkmale, wie Datenmodell, Transaktionen und Versionsverwaltung genauer eingegangen. Weiterhin werden in Kapitel

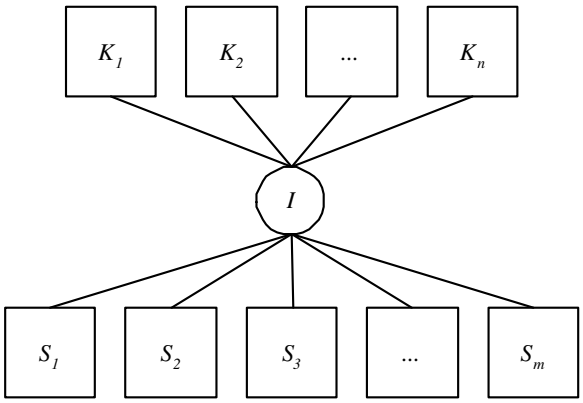

Abbildung 2: Integration von Informationssystemen mit einheitlicher Schnittstelle

2 einige Programmierschnittstellen für Informationssysteme vorgestellt. Kapitel 3 geht im Detail auf die *Java Content Repository* API ein, da diese Schnittstelle alle der in Kapitel 2 gefundenen Anforderungen erfüllt und für die in Kapitel 5 vorgestellte Implementierung der Integration in die CoreMedia Smart Content Technology verwendet wird. Kapitel 4 erklärt die wichtigsten Komponenten dieser Infrastruktur. In Kapitel 6 werden die Ergebnisse der Arbeit zusammengefasst.

2 Integration von heterogenen Informationssystemen

Um verschiedene Informationssysteme in unterschiedliche Klienten zu integrieren, muss zunächst der Begriff „Informationssystem“ genauer erklärt werden. Außerdem müssen die Dienste, welche so ein Informationssystem bietet, betrachtet werden. Anschließend werden allgemeine Schnittstellen für Informationssysteme untersucht.

2.1 Klassifikation von Informationssystemen

Informationssysteme, so wie sie im vorherigen Kapitel charakterisiert wurden, lassen sich in verschiedene Klassen einteilen. Die Basis für alle Informationssysteme sind die Betriebssysteme. Darauf aufbauend wurden verschiedene Systeme entwickelt, die speziellere Funktionalität bieten und die Dienste generellerer Systeme nutzen (siehe Abbildung 3).

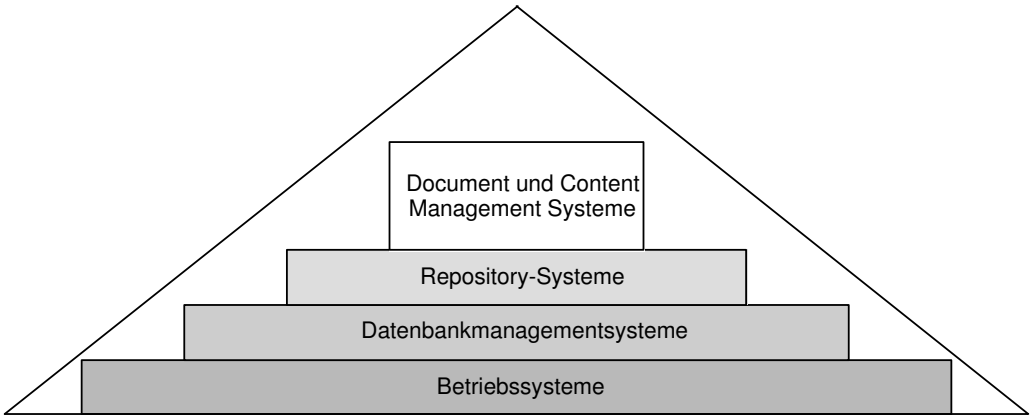

Abbildung 3: Klassifikation von Informationssystemen

Die Betriebssysteme bieten sehr allgemeine Dienste, wie Dateisysteme, welche die Persistenz von Daten sicherstellen. Weiterhin stellen Betriebssysteme grundlegende Dinge zur Verfügung, wie Hardware-Abstraktion oder Netzwerkkommunikation.

Darauf aufbauend wurden Datenbankmanagementsysteme entwickelt. Diese bieten neben der Datenpersistenz auch Datenintegrität, Transaktionen und Datendefinitions- und manipulationssprachen.

Repository-Systeme bieten erweiterte Dienste, wie objektorientierten Datenbankzugriff, Versionsverwaltung und Überwachung von Datenobjekten.

Den Dokumentmanagement und Content Management Systeme liegt in der Regel ein Repository-System zu Grunde. Sie sind Instanzen dieser Systeme und erweitern deren Funktionalität um die Unterstützung für den Lebenszyklus von Informationsobjekten und Workflowmanagement für den Arbeitsablauf.

Im Folgenden werden Repository-Systeme als spezielle Form eines Informationssystems betrachtet. Wird von Informationssystemen gesprochen, sind Repository-Systeme gemeint. Es wird versucht für diese Klasse von Systemen eine Schnittstelle zu finden, die eine einheitliche Kommunikation von Klienten mit dieser Klasse von Systemen erlaubt.

2.1.1 Abgrenzung

Die hier vorgestellte Repository-Technologie steht im Gegensatz zu anderen Technologien zur Integration von Informationssystemen. Diese Arbeit bezieht sich auf die Integration von operativen Informationen innerhalb eines Unternehmens und auch zwischen Unternehmen. Integriert werden hierbei Informationen wie Produkt- oder Kontaktdaten. Diese Art der Integration steht im Kontrast zu anderen Integrationstechnologien, wie *Enterprise Application Integration* oder *Data Warehousing*. Die Unterschiede werden in den folgenden Abschnitten erläutert.

Enterprise Application Integration

Enterprise Application Integration, kurz *EAI*, ist eine Technologie, die benutzt wird um mehrere Anwendungen innerhalb eines Unternehmens miteinander kommunizieren zu lassen, damit sie Informationen und Funktionalität wieder verwenden können. Dadurch lassen sich Geschäftsabläufe automatisieren.

Ein Hauptprinzip von *EAI* ist die Integration von Insellösungen ohne dabei existierende Altsysteme, die zufrieden stellend arbeiten, ersetzen zu müssen. Um dieses bewerkstelligen zu können, enthält eine *EAI*-Architektur Anwendung-zu-Anwendung-Adapter, Geschäftsregeln, Daten-Transformations-Technologien und Workflow Management.

Im Gegensatz zu EAI beschäftigt sich die in dieser Arbeit vorgestellte Art der Integration nicht mit der Integration von Geschäftsprozessen.

Data Warehousing

Operative Systeme helfen das Tagesgeschäft eines Unternehmens zu führen. Sie sind das Rückgrat jedes Unternehmens und beinhalten Tätigkeiten wie „Auftragsannahme“, „Vorratshaltung“, „Gehaltsabrechnung“ und „Buchhaltung“. Performanz ist eine der Hauptanforderungen an operative Systeme. Daher ist es nicht praktikabel Daten unendlich lange in operativen Systemen zu halten. Aus diesem Grund werden so genannte Data Warehousing Systeme verwendet. In diesen Systemen sind Daten untergebracht, die aus operativen Systemen stammen und dort, sofern nicht mehr benötigt, aus Performanz-Gründen entfernt worden sind. Um die Daten für Analysezwecke aufzubewahren, werden sie in Data Warehouse Systeme kopiert.

Analysesysteme, wie Data Warehouse Systeme, werden zur Analyse von Daten und als Entscheidungshilfen genutzt. Sie arbeiten mit historischen Daten, die über die Zeit in operativen Systemen angefallen sind und dort aus Performanz-Gründen aber nur begrenzt gespeichert werden können. Im Gegensatz zu operativen Systemen, die in der Regel auf eine

Domäne, wie zum Beispiel „Gehaltsabrechnung“ spezialisiert sind, beinhalten Analysesysteme oft große Mengen historischer Daten aus unterschiedlichen Domänen.

Die in dieser Arbeit gezeigte Art der Integration dient zur unternehmensweiten oder unternehmensübergreifenden Integration von operativen Daten und unterscheidet sich damit vom klassischen *Data Warehousing*.

Mediatoren

In [Wie97] wird ein ähnlicher Ansatz verfolgt wie in dieser Arbeit. Jedoch geht der dort vorgestellte Ansatz insofern weiter, als das mehrere Informationssysteme durch einen so genannten Mediator gekapselt werden. Ein Mediator fasst die integrierten Systeme in einem einzigen Informationsmodell zusammen. So ist es zum Beispiel möglich eine Anfrage an den Mediator zu stellen, der diese dann an die unterschiedlichen integrierten Systeme verteilt und die Ergebnisse anschließend wieder zusammenfasst.

2.2 Dienste von Informationssystemen

Informationssysteme bieten und nutzen eine Reihe von Diensten. Einige von diesen Diensten werden durch die Informationssysteme selbst implementiert. Für andere Dienste werden Systeme aus anderen Schichten des Anwendungsstapels (siehe Abbildung 4) benutzt. Das Informationssystem als eigentliche Anwendung ist das oberste Element dieses Stapels. Es nutzt Dienste aus darunter liegenden Schichten. Für Datenbankfunktionalität wird in der Regel ein Datenbankmanagementsystem (DBMS) verwendet, welches in der Middleware-Schicht angesiedelt ist. Diese Schicht bietet erweiterte Dienste zur Nutzung durch Anwendungssysteme. Es nutzt, wie das Anwendungssystem selbst, grundlegende Dienste, die vom Betriebssystem bereitgestellt werden. Dazu zählen Dateisysteme oder Netzwerkkommunikationsdienste.

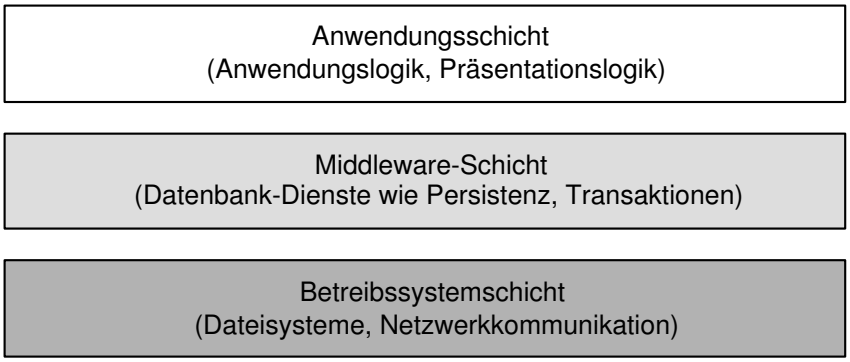

Abbildung 4: Mehrschicht-Architektur von Informationssystemen

Wie im vorherigen Abschnitt angedeutet, werden häufig benutzte Dienste durch die Informationssysteme selbst implementiert. Dazu gehört zum Beispiel Versionsverwaltung und

Überwachung. Diese werden von vielen Software-Konfigurations- sowie Content Management und Dokumentenmanagement Systemen, aufbauend auf Datenbanktechnologie, realisiert.

Die Entwicklung in der Informationstechnologie zeigt, dass häufig genutzte Dienste nach einiger Zeit aus der Anwendungsschicht in die Middleware-Schicht und noch später in die Betriebssystemschicht wandern und als eigenständiges Informationssystem bzw. Produkt für diese Schicht angeboten werden. So ist die Datenbanktechnik, welche vor 25 Jahren durch die Anwendungsschicht realisiert wurde, heute als DBMS in der Middleware-Schicht anzutreffen. Und die neuere Entwicklung anhand von Microsoft's WinFS [Gri04] zeigt, dass die Datenbanktechnik demnächst in die Betriebssystemschicht integriert wird.

Andere Kandidaten für einen „Schichtwechsel" sind die Dienste, welche heute von vielen Dokumenten- und Content Management Systemen realisiert werden. Diese Dienste werden als Repository-Technologie bezeichnet. Im Folgenden werden einige Dienste und Merkmale von Repository-Systemen näher betrachtet, die in [Ber97] identifiziert wurden. Es wird untersucht, in wie weit es für die einzelnen Dienste schon standardisierte Schnittstellen gibt, die sich in eine einheitliche Schnittstelle für Repository-Systeme integrieren ließen.

2.2.1 Datenmodelle und Typsysteme

Dieser Abschnitt gibt einen Überblick über die für die Repository-Technologie interessanten und relevanten Datenmodelle.

Relationales Datenmodell

Im Bereich der Datenbanken spielt das relationale Modell [Cod70] aufgrund seiner weiten Verbreitung eine wichtige Rolle. Das Modell kennt ausschließlich Tabellen als Datenstruktur, die aus Zeilen und Spalten bestehen und Daten von skalaren Typen enthalten.

Die Standard-Datenbanksprache bei relationalen Datenbanksystemen ist SQL [ISO92]. Sie umfasst Konstrukte zur Datendefinition und -manipulation sowie zur deskriptiven Formulierung von Anfragen.

Die wichtigen Konzepte des relationalen Datenmodells sind:

- **Datenbank**
  Eine Datenbank besteht aus einer Menge von Relationen.
- **Relation**
  Eine Relation ist eine Menge von Elementen, deren Struktur durch Attribute definiert, deren Identität durch Schlüssel realisiert und deren Werte durch Domänen kontrolliert werden.
- **Tabellen**
  Relationen werden als Tabellen dargestellt, die aus Zeilen und Spalten bestehen.
- **Tupel**
  Jede Zeile in einer Tabelle repräsentiert ein Element der Relation, das auch Tupel genannt wird.

- **Attribute**
  Die Spalten der Tabellen enthalten die Attribute der Relation.
- **Domäne**
  Jede Spalte einer Tabelle ist eine Domäne zugeordnet, die festlegt, welche Werte für die Attribute in dieser Spalte gültig sind.
- **Kardinalität**
  Die Anzahl der Zeilen einer Tabelle ist variabel und wird Kardinalität genannt.
- **Grad der Relation**
  Die Anzahl der Spalten einer Tabelle wird Grad der Relation genannt.
- **Primärschlüssel**
  Ein Primärschlüssel ist ein ausgezeichnetes Attribut oder eine Attributkombination, die eine eindeutige Identifikation eines Tupels innerhalb einer Relation ermöglicht.
- **Fremdschlüssel**
  Da das relationale Modell keine Referenzen kennt, werden Beziehungen zwischen Datenobjekten über den Primärschlüssel des referenzierenden Objekts hergestellt. Dieser Schlüssel wird Fremdschlüssel genannt.

Objekt-orientiertes Datenmodell

Im Gegensatz zum relationalen Modell, wo sich ein Standard entwickelt hat, gibt es bei den objektorientierten Datenbanken unterschiedliche Ansätze von einigen Herstellern kommerzieller Systeme. Den wichtigsten Beitrag zur Standardisierung hat die *Object Data Management Group* (ODMG) geliefert. Sie hat ODMG 3.0 [CBB+00] als Vorschlag für ein objektorientiertes Datenbanksystem (OODBMS) veröffentlicht. Dieser beinhaltet ein Objektmodell, eine Datendefinitionssprache, eine Anfragesprache, ein Datenaustauschformat und Sprachbindungen für diverse objektorientierte Programmiersprachen.

Bei objektorientierten Datenbanksystemen findet eine implizite Kommunikation zwischen Anwendungsprogrammen und dem Datenbanksystem statt. Der Programmierer arbeitet direkt mit den persistenten Datenstrukturen, die dem gleichen Typsystem folgen, wie die Strukturen im flüchtigen Hauptspeicher, sodass hier keine explizite Konvertierung stattfinden muss. Die Programmiersprache, die zur Erstellung des Anwendungssystems genutzt wird ist auch gleichzeitig Datendefinitions- und manipulationssprache für die objektorientierte Datenbank.

Ein objektorientiertes Datenmodell bietet zudem mächtigere Modellierungsmöglichkeiten als das relationale Modell. Anwendungsnahe Konzepte, wie Klassen, Nachrichten und Objekte lassen sich direkt auf Informationssysteme übertragen.

Zu den wichtigen Konzepten objektorientierter Datenbanken gehören neben algorithmischer Vollständigkeit, Persistenzabstraktion, Sekundärspeicherverwaltung, Mehrbenutzerbetrieb und Fehlererholung:

- **Typkonstruktoren und komplexe Objekte**
  Mit der Hilfe von Konstruktoren können komplexe Objekte aus einfacheren Objekten und

den Basistypen wie *Integer* oder *Boolean* aufgebaut werden. Ein objektorientiertes Datenmodell muss mindestens Konstruktoren für Aggregattypen, Mengen und geordneten Kollektionen bieten. Diese sollten auf jeden Datentyp anwendbar sein.

- **Objektidentität**
  Jedem Objekt wird bei seiner Erzeugung ein Objektidentifikator zugewiesen, der unabhängig von den Werten des Objekts ist, systemweit eindeutig, unabhängig vom Zustand des Objekts und über die gesamte Lebensdauer des Objekts gleich ist. Diese Identifikatoren werden vom System zugewiesen und können nicht vom Benutzer manipuliert werden.

- **Kapselung durch Methoden**
  Auf den Zustand eines Objekts kann nur über Methoden zugegriffen werden. Diese Kapselung stellt sicher, dass Integritätsprüfungen und Zugriffskontrollen im Rahmen der Methodenaufrufe durchgeführt werden.

- **Typen und Klassen**
  Die Struktur von Objekten wird von Typen festgelegt. Diese legen auch die Signaturen von Methoden und die zulässigen Wertebereiche fest. Eine Klasse definiert Attribute, Methoden eines Typs. Eine Klasse definiert einen Typ.

- **Vererbung**
  Klassen können in Hierarchien angeordnet werden. Klassen die tiefer in der Hierarchie stehen (Subklassen) erben die Attribute und Methoden von Klassen die höher in der Hierarchie stehen (Superklassen). Die Vererbung unterstützt die Wiederverwendung von Definitionen und Implementierungen. Jedes Objekt einer Subklasse ist auch eine Instanz aller Superklassen.

- **Methodenredefinition und späte Bindung**
  Für jede Methode einer Klasse kann es unterschiedliche Implementierungen geben (*Overloading*). Diese Methoden haben dann leicht unterschiedliche Signaturen. In Subklassen können Methoden jedoch auch überschrieben, dass heißt neu definiert, werden (*Overwriting*). Welche Implementierung einer Methode beim Aufruf ausgeführt wird, hängt von der Klasse eines Objekts ab, dessen Methode aufgerufen werden soll. Dazu wird die Klassenhierarchie nach oben hin durchsucht und die zuerst gefundene Implementierung ausgeführt.

- **Assoziationen zwischen Klassen**
  Assoziationen zwischen Klassen werden als Attribute der assoziierenden Klasse dargestellt. Der Typ eines solchen Attributs ist die assoziierte Klasse. Mengenwertige Assoziationen werden durch Mengentypen realisiert.

Extensible Markup Language

XML [BPSMM00] definiert neben einem Serialisierungsformat auch ein abstraktes Datenmodell. Dieser Abschnitt gibt einen Überblick über das strukturelle Modell eines XML Dokuments,

wie in der *XML Information Set* (InfoSet) Spezifikation [CT01] beschrieben, über XML Schema zur Typisierung solcher Dokumente und über die Anfragesprachen XPath und XQuery.

Die Teile eines XML Dokuments werden durch InfoSet als eine Menge von Informationselementen (*information items*) abstrahiert, die eine oder mehrere benannte Eigenschaften besitzen. InfoSet ist nicht an die Syntax gebunden, die durch [BPSMM00] beschrieben wird.

InfoSet definiert verschiedene *information items*, die den einzelnen Entitäten eines XML Dokuments entsprechen. Das *document information item* ist die Wurzel des Objektmodells, welches eine Baumstruktur besitzt. *Element information items* sind Knoten in diesem Baum und *Processing instruction*, *comment* und *character information items* sind Blätter. Dieser Graph wird über zwei spezielle Eigenschaften aufgebaut. Alle *information items* (außer dem *document information item*) besitzen eine `parent`-Eigenschaft, die den Vaterknoten referenziert. Außer den Blattelementen besitzt jedes Element eine `children` Eigenschaft, die eine geordnete Liste aller direkten Kindknoten enthält. Weiterhin hat jedes *element information item* eine `attributes` Eigenschaft, die eine Liste von ungeordneten, eindeutig benannten *attribute information items* darstellt.

Mit Hilfe der *element information items* können hierarchische Datenstrukturen abgebildet werden. Ein Element, dass einen Basistypen abbildet, enthält nur Zeichen (*character information items*). Elemente, die strukturierte Typen repräsentieren, enthalten in der Regel keine Zeichen sondern Kindelemente. *Attribute information items* dienen typischerweise als Name-Wert-Paare.

Um die Struktur von XML Dokumenten zu beschreiben wurde die XML Schema Spezifikation [Fal01, TBMM01, BM01] verabschiedet. Im Gegensatz zu DTDs basiert eine XML Schema Definition auf Typen und nicht auf Elementnamen. Dadurch lassen sich Strukturen aus Programmiersprachen oder Datenbanken leichter abbilden. XML Schema Definitionen sind ebenfalls XML Dokumente und können so mit vorhandener XML Technologie verarbeitet werden.

Um Anfragen an XML Dokumente zu stellen wurden einige Anfragesprachen definiert. Zwei der interessanten davon sind XPath [CD99] und XQuery [BCF+03].

XPath ist eine Sprache, die dazu dient Teile eines XML-Dokuments zu adressieren. Sie wurde ursprünglich für die Verwendung in XSLT [Cla99] und XPointer [GMMW03] entworfen. Um den primären Zweck der Adressierung eines XML-Fragments zu unterstützen, bietet XPath einfache Möglichkeiten um Zeichenketten, Zahlen und Boolsche-Werte zu manipulieren. Um XPath-Ausdrücke in URIs und XML Attributwerten verwenden zu können, benutzt XPath eine kompakte Syntax, die nicht auf XML basiert. Die Sprache wird XPath genannt, da sie eine Pfad-Notation, ähnlich wie in URLs, benutzt, um durch die hierarchische Struktur eines XML-Dokuments zu navigieren. XPath arbeitet mit der logischen Struktur eines XML-Dokuments anstatt mit seiner serialisierten Form.

XQuery ist eine Sprache, die entworfen wurde, um Anfragen einfach und übersichtlich ausdrücken zu können und so vom menschlichen Leser leicht verstanden werden kann. XQuery wurde von Quilt [CRF00], einer anderen XML-Anfragesprache angeleitet, die ihrerseits einige Merkmale von anderen Sprachen wie XPath 1.0 [CD99], XQL [RLS98], XML-QL [DFF+98], SQL [ISO92] und OQL [CBB+00] enthält. XQuery operiert auf der abstrakten, logischen

Struktur von XML Dokumenten, und nicht auf seiner Syntax. Diese logische Struktur wird als Datenmodell bezeichnet und ist in [FMM+03] beschrieben. Das Typsystem von XQuery basiert auf XML Schema. XQuery in der Version 1.0 ist eine Erweiterung von XPath Version 2.0.

Der Grundbaustein von XQuery ist ein Ausdruck, der aus einer Kette von Unicode-Zeichen besteht. Die Sprache definiert mehrere Arten von Ausdrücken, die aus Schlüsselwörtern, Symbolen und Operanden aufgebaut werden. Operanden können im Allgemeinen wiederum Ausdrücke sein. XQuery ist eine funktionale und stark-typisierte Sprache. Dadurch können Ausdrücke andere Ausdrücke enthalten und die Operanden der verschiedenen Ausdrücke, Operatoren und Funktionen müssen dem erwarteten Typ entsprechen.

2.2.2 Transaktionsverarbeitung

Die Verarbeitung von Transaktionen ist ein wichtiges Merkmal von Informationssystemen. Ohne Transaktionen ist eine verteilte Informationsverarbeitung mit mehreren beteiligen Systemen nicht vorstellbar. Der ersten Transaktionsmonitore, wie CICS von IBM, wurden in den sechziger Jahren des 20. Jahrhunderts entwickelt. In der Mitte der achtziger Jahre des 20. Jahrhunderts begann man mit der Entwicklung offener Standards im Bereich Transaktionsverarbeitung. Heute unterstützen viele Produkte, die an der Verarbeitung von Transaktionen beteiligt sind, einige dieser Standards.

Da in dieser Arbeit nach einer einheitlichen Schnittstelle für Informationssysteme gesucht wird, werden im Folgenden nur die offen, standardisierten Transaktionsmodelle und deren Schnittstellen vorgestellt.

OSI TP

Das OSI TP (*Open Systems Interconnect Transaction Processing*) Protokoll wurde von der ISO (*International Standards Organization*) entwickelt. Es ist das einzige Protokoll im Bereich der Transaktionsverarbeitung, das von einem unabhängigen Standardisierungsgremium entwickelt wurde.

Dieses Protokoll fand jedoch kaum Unterstützung bei den Herstellern von transaktionsverarbeitenden Systemen. Einige Teile des X/Open Modells basieren jedoch auf OSI TP.

In letzter Zeit wurden einige neue Beiträge bei der ISO eingebracht, die als ISO/IEC 10026 [ISO98] verabschiedet wurden und die Standardisierung von Schnittstellen und Modellen zur Transaktionsverarbeitung weiter voranbringen sollen. Insbesondere wurde auch die X/Open Spezifikation (siehe unten) im beschleunigten Verfahren als ISO/IEC 14834 [ISO96a] übernommen.

OMG Object Transaction Service

Die *Object Management Group* (OMG) ist ein Industriekonsortium, welches die *Common Object Request Broker Architecture* (CORBA) Spezifikation [COR02] entwickelt. CORBA ist eine Schnittstelle zum Aufruf von entfernten Objekten (*remote object invocation*). OMG hat die *Object Transaction Service* (OTS) [OTS03] Spezifikation publiziert, welche

einen transaktionalen Dienst für eine objektorientierte Programmierumgebung definiert, der auf CORBA basiert. Diese Spezifikation baut auf der X/Open Spezifikation auf und sichert so die Kompatibilität zwischen dem OTS und dem X/Open Modell für verteilte Transaktionsverarbeitung.

Spezifikationen der OMG erlauben einen hohen Grad an Flexibilität bei der Interpretation und sehen viele optionale Teile vor. Dadurch variiert die Unterstützung von OTS von Hersteller zu Hersteller.

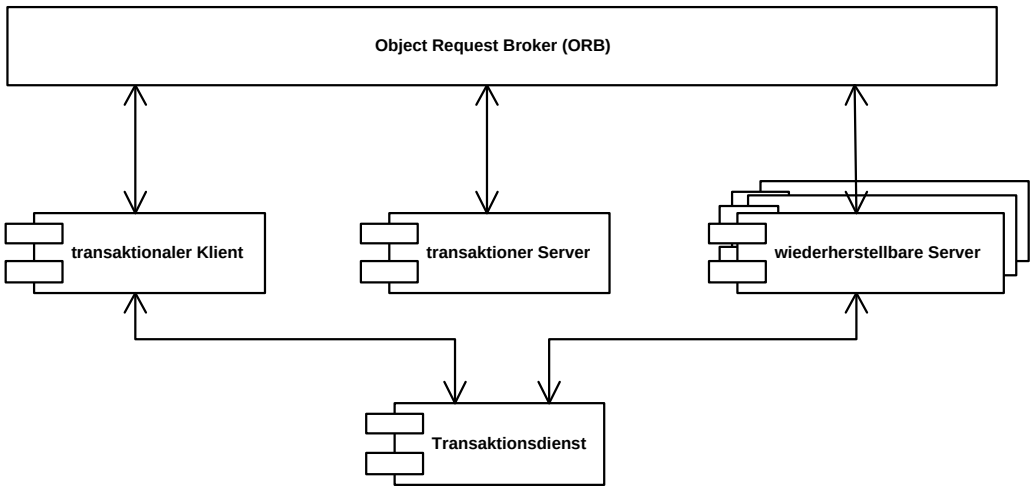

Abbildung 5: Der Object Transaction Server der OMG

Wie Abbildung 5 zeigt, sieht das OTS Modell einen transaktionalen Klienten, einen transaktionalen Server und wiederherstellbare Server, die sich von Fehlern erholen können, vor.

Der größte Unterschied zwischen dem objektorientierten OTS Modell und dem prozeduralen X/Open Modell besteht darin, dass alle Transaktionsoperationen als Methodenaufrufe auf Objekten durchgeführt werden.

Eine Transaktion wird von dem Transaktionsdienst gestartet und kontrolliert. Die transaktionalen Klienten und Server kommunizieren durch den *Object Request Broker* (ORB), der den Transaktionskontext vom transaktionalen Klienten zum transaktionalen Server und zu den wiederherstellbaren Servern überträgt. Optional kann der Kontext auch explizit vom Anwendungsprogramm übertragen werden.

Der ORB erfüllt viele der Funktionen eines Transaktionsmonitors. Er ist ein Modell für das Verhalten eines Transaktionsmonitors in der Objektarchitektur der OMG.

Im OTS Modell wird ein Objekt über ein Attribut der CORBA Schnittstellenbeschreibungssprache (*Interface Definition Language, IDL*) als transaktional markiert. Wird eine Methode eines transaktionalen Objekts von einem transaktionalen Klienten aufgerufen, leitet der ORB

den Aufruf an den transaktionalen Server weiter. Der ORB verwaltet den Transaktionskontext mit Hilfe des Transaktionsdienstes.

Eine Implementierung des OTS erfordert kein 2-Phasen-Commit Protokoll. Dadurch hängt die Interoperabilität des Dienstes von den Herstellern der Transaktionskomponenten ab.

Das X/Open Modell

Das Ziel von X/Open ist es, die Portabilität von Anwendungen durch die Standardisierung von Programmierschnittstellen zu erreichen. X/Open hat ein Modell zur verteilen Transaktionsverarbeitung (*distributed transaction processing, DTP*) entwickelt [XA92], welches viele der grundlegenden Funktionen enthält, die von Transaktionsmonitoren angeboten werden. Die erste Version wurde 1991 veröffentlicht und beschrieb Transaktionsabgrenzung sowie Schnittstellen von Komponenten die Ressourcen verwalten (*resource managers*) zu Komponenten die Transaktionen verwalten (*transaction managers*). Das erste Modell wurde ständig weiterentwickelt und enthält mittlerweile viele Kommunikationsschnittstellen und eine Sprache zur Kontrolle von Arbeitsabläufen.

Das X/Open Modell zur verteilten Transaktionsverarbeitung teilt ein transaktionsverarbeitendes System in Komponenten auf und definiert Schnittstellen zwischen diesen. Zu den Komponenten zählen *Transaction Manager*, Datenbanken oder *Resource Manager* und *Transactional Communications Manager*. Abbildung 6 zeigt dieses Modell.

Die Hauptkomponenten und -schnittstellen des Modells sind:

- **TX**
  Definiert die Schnittstelle zwischen dem Anwendungsprogramm und dem *Transaction Manager*. Sie wird benötigt um Transaktionen zu starten und zu beenden sowie um Statusinformationen über eine Transaktion abzufragen.

- **XA**
  Definiert die Schnittstelle zwischen einem *Resource Manager* und dem *Transaction Manager*. Wird die Schnittstelle von einem Transaktionsmonitor und einer Datenbank unterstützt, können beide zusammen eine Transaktion zwischen ihnen koordinieren. Die XA Schnittstelle hat breite Industrieunterstützung, wodurch es möglich ist viele Transaktionsmonitore und Datenbanken zu kombinieren.

- **CRM**
  Ein *Communications Resource Manager* biete eine Programmierschnittstelle auf Anwendungsebene zu einem Kommunikationsprotokoll für entfernte, transaktionale Kommunikation.

- **XA+**
  Die XA+ erweitert XA um eine Schnittstelle zwischen einem *Transaction Manager* und einem *Communications Resource Manager*, sodass der CRM dem *Transaction Manager* mitteilen kann, dass ein ihm untergeordneter Knoten an der verteilten Transaktion teilnimmt.

- **RM**
  Die Schnittstelle zu einem *Resource Manager*, zum Beispiel einer Datenbank, wird von der X/Open Spezifikation nicht festgeschrieben. Ein *Resource Manager* sollte aber wiederherstellbar sein und sich von Fehlerfällen erholen können.

- **XAP-TP**
  Die Programmierschnittstelle zwischen einem *Communications Resource Manager* und OSI TP.

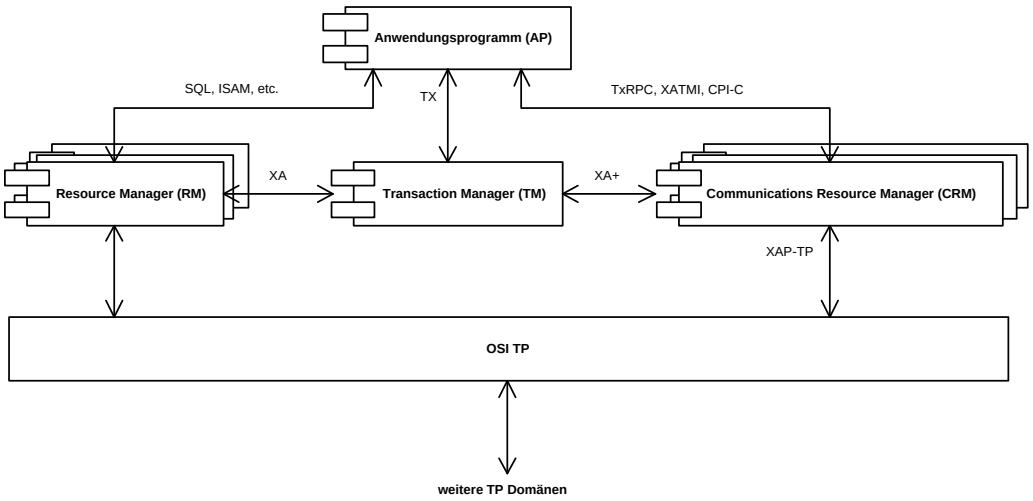

Abbildung 6: Das X/Open Modell zur verteilten Transaktionsverarbeitung

Für die XA Schnittstelle existiert eine Abbildung auf eine Programmierschnittstelle für die Java 2 Plattform, die Java Transaction API (JTA) [JTA99]. Die Schnittstelle wird von anderen APIs, wie JDBC [EH01] und der Java 2 Enterprise Edition (J2EE) [Sha01] benutzt und wird von vielen Herstellern unterstützt. Sie definiert Java Schnittstellen zwischen einem *Transaction Manager* und anderen Beteiligten an verteilten Transaktionen, wie dem Anwendungsprogramm, dem *Resource Manager* und dem J2EE Anwendungsserver. Dadurch ist eine hohe Interoperabilität in Bezug auf Transaktionen zwischen Java-Anwendungen und -Servern gegeben.

Da in dieser Arbeit nach einer allgemeinen Java Schnittstelle für Informationssysteme gesucht wird und diese auch Transaktionen unterstützen muss, erscheint es sinnvoll die JTA API in diese Schnittstelle einzubeziehen.

2.2.3 Versionsverwaltung

Informationen, die in einem *Repository* gespeichert werden, haben in der Regel eine längere Lebenszeit als die Anwendungen, die diese Informationen erzeugt haben. Während

einige Daten, nachdem sie gespeichert wurden, nie geändert werden, unterliegen andere häufigen Änderungen. Viele Datenbankanwendungen benötigen lediglich den aktuellen Zustand eines Objektes. Bei einer Änderung wird der alte Zustand mit dem Neuen überschrieben. Es existieren jedoch Anwendungen, wie zum Beispiel im Bereich des *Computer-Aided Design* (CAD) oder des *Software Configuration Management* (SCM), welche die Anforderung besitzen auf die alten Zustände eines Informationsobjekts zugreifen zu müssen. Diese unterschiedlichen Zustände eines Informationsobjekts werden Versionen genannt.

Um die Anforderungen solcher Anwendungen abzudecken, sollte eine allgemeine *Repository*-Programmierschnittstelle eine (optionale) Schnittstelle zu versionierten Informationen bieten. Eine solche Schnittstelle sollte die zwei Schlüsselkonzepte *Workspaces* und Versionierung abdecken.

Ein *Workspace* stellt eine Umgebung zur Verfügung, in der ein Benutzer persistente Änderungen an Ressourcen vornehmen kann ohne auf andere Benutzer störend einzuwirken oder von diesen beeinflusst zu werden. Dieses erfordert, dass die Änderungen des Benutzers persistent gespeichert werden können, ohne dass dessen Änderungen sofort für alle anderen Benutzer sichtbar sind. Um dieses zu bewerkstelligen, werden die Änderungen zunächst in einem *Workspace* gespeichert und müssen von dem Benutzer explizit für die Verwendung in anderen *Workspaces* herausgestellt werden. Damit ein anderer Benutzer diese Änderungen in seinem *Workspaces* sieht, muss er sie explizit anfordern.

Versionierung bietet eine Umgebung, in der vorhergehende Zustände von Ressourcen leicht zugreifbar sind. Konfigurationsverwaltung erweitert dieses Konzept dahingehend, dass nicht nur die Zustände einzelner Ressourcen zugreifbar sein müssen, sondern konsistente Konfigurationen von verwandten Ressourcen.

Zu den Vorteilen von Versions- und Konfigurationsverwaltung gehören:

- Eine Ressource hat eine explizite Historie und eine eindeutige Identität während der Zustandsänderungen entlang dieser Historie. Es ist möglich zu älteren oder alternativen Versionen einer Ressource zu navigieren.
- Den Zuständen von Ressourcen werden Namen zugewiesen, sodass auf einen Zustand verwiesen werden kann.
- Eine Menge von Ressourcen kann parallel geändert werden. Jeder Benutzer kann Änderungen durchführen ohne andere Benutzer zu stören. Konsistente Mengen von Änderungen können dann zu anderen Benutzern übertragen werden.

Es existieren viele Versionierungsmodelle, die für *Software Configuration Management* und objektorientierte Datenbanken entworfen wurden [CW98, ABGS91, CW97, Kat90, KCB86]. Es hat sich hieraus jedoch kein einheitliches Modell und damit auch keine einheitliche Schnittstelle zu versionierten Objekten entwickelt.

Zwei für den Bereich der Repository-Technologie interessante Bemühungen werden im Folgenden vorgestellt.

Versioning Extensions to WebDAV (DeltaV)

WebDAV [Whi98, WW98] (*Web Distributed Authoring and Versioning*) ist eine Erweiterung des HTTP-Protokolls [FGM+99]. Trotz seines Namens definiert WebDAV nur Möglichkeiten zum kollaborativen, entfernten Bearbeiten von Dokumenten.

Die Erweiterungen umfassen die folgenden Konzepte:

- **Eigenschaften**
  Eigenschaften erlauben es Meta-Daten, wie Erstellungsdatum oder Autor, für Web-Ressourcen zu erzeugen, zu löschen und abzufragen.
- **Sammlungen**
  Sammlungen sind Mengen von Web-Ressourcen. Sammlungen können weitere Sammlungen enthalten.
- **Sperren**
  Sperren erlauben das exklusive Bearbeiten einer Web-Ressource durch genau einen Benutzer.
- **Operationen auf Namensräumen**
  Diese Operationen umfassen das Kopieren und Verschieben von Web-Ressourcen.

DeltaV [CAE+02] erweitert WebDAV um die fehlenden Versionierungsmerkmale und beschreibt ein Modell und Protokoll um Web-Ressourcen unter Versionskontrolle zu stellen. Da der Fokus dieser Arbeit auf der Suche nach einer Programmierschnittstelle liegt und da das semantische Modell von DeltaV mit dem der *Workspace Versioning and Configuration Management API* (siehe unten) kompatibel ist, wird hier auf eine detaillierte Beschreibung von DeltaV verzichtet.

Workspace Versioning and Configuration Management API

Dieser Abschnitt gibt einen Überblick über die *Workspace Versioning and Configuration Management (WVCM) API* [WVC03].

Die Programmierschnittstelle soll die Komplexität von Anwendungsprogrammen reduzieren, die mit versionierten und nicht-versionierten Repositories kommunizieren müssen. Die Schnittstellendefinition beschreibt grundlegendes *Workspace Management*, die Verwaltung von Versionshistorien, Grundlinien und die Versionierung von Namensräumen.

Um die Interoperabilität zu erhöhen und existierende Protokolle auszunutzen, ist das semantische Modell der WVCM API kompatibel mit DeltaV, WebDAV und HTTP.

Bei dem WVCM-Modell enthält ein Repository eine Menge von Ressourcen. Eine Ressource ist ein auffindbares, persistentes Objekt, dessen Zustand aus „Inhalt“ und einer Menge von „Eigenschaften“ (Name-Wert-Paaren) besteht. Eine „Ordner Ressource“ oder kurz „Ordner“ ist eine Ressource, deren Zustand eine Menge von Bindungen ist. Eine Bindung ist die Abbildung von einem Namen auf eine andere Ressource, die auch „gebundenes Mitglied“ genannt wird.

Die Semantik dieses Modells sieht wie folgt aus:

- Erzeugen einer versionierten Ressource

  Um die Historie über den Inhalt einer Ressource aufzuzeichnen, kann eine versionierbare Ressource unter Versionskontrolle gestellt werden. Daraufhin werden die folgenden Operationen ausgeführt:

  - Es wird eine neue Versionshistorie erzeugt. Eine Versionshistorie ist eine Ressource, die alle Versionen enthält, die Nachfolger einer bestimmten Version sind. Diese Version ist der Wurzelversion dieser Versionshistorie.
  - Es wird eine neue Versionsressource angelegt und diese zu der Versionshistorie hinzugefügt. Der Inhalt dieser Ressource ist eine Kopie der versionierbaren Ressource, die unter Versionskontrolle gestellt wird. Eine Versionsressource enthält also eine Kopie des Zustands einer versionierten Ressource.
  - Die versionierbare Ressource wird eine versionierte Ressource.

- Modifizieren einer versionierten Ressource

  Im den Inhalt einer versionierten Ressource zu verändern, muss diese zuerst ausgeliehen werden (*check-out*). Wenn eine ausgeliehene Ressource zurückgegeben wird (*check-in*), wird eine neue Versionsressource erzeugt und in die Versionshistorie eingetragen. Die Versionsressource der Version, die ausgeliehen wurde, wird als Vorgänger der neuen Versionsressource vermerkt.

- Versionierte Ordner

  Wird ein Ordner unter Versionskontrolle gestellt, wird ebenfalls eine Versionshistorie angelegt. Die Versionsressourcen eines Ordners enthalten Informationen über die gebundenen Mitglieder des Ordners. Um zwischen Änderungen am Namensraum und Änderungen am Inhalt zu unterscheiden, enthält die Versionsressource eines Ordners den Bindungsnamen und die Versionshistorie jedes gebundenen Mitglieds. Dadurch ist es nicht notwendig eine neue Version eines Ordners anzulegen, wenn eine andere Version eines gebundenen Mitglieds ausgewählt wird.

- Konfigurationen

  Eine Konfiguration ist eine Menge von Ressourcen, die alle Mitglieder eines Wurzelordners enthält, die nicht in anderen Konfigurationen enthalten sind.

- Grundlinien

  Eine Grundlinie (*baseline*) ist eine Versionsressource, die den Zustand eines jeden versionierten Mitglieds einer Konfiguration enthält. Eine Grundlinienhistorie ist eine Versionshistorie, deren Versionen Grundlinien sind. Neue Grundlinien werden erzeugt, indem spezielle versionierte Ressourcen, versionierte Konfigurationen genannt, ausgeliehen und zurückgegeben werden.

2.2.4 Überwachung

Da Anwendungen die Daten in einem Informationssystem teilen, also auf eine gemeinsame Datenbasis zugreifen, kann es notwendig sein, die Anwendungen zu benachrichtigen, wenn spezifische Operationen, wie Änderungen, an Daten vorgenommen werden, die für die Anwendung relevant sind.

Diese Benachrichtigungen werden zu den Anwendungen in Form von Ereignissen übermittelt. Zu diesem Zweck wurden Ereignis-Architekturen, wie [MB], Ereignis-Dienste, wie der von der OMG spezifizierte *CORBA Event Service* [Obj01] und Nachrichten-Dienste, wie der *Java Messaging Service* (JMS) [HBS+02] entworfen. Ereignis- oder Nachrichten-Dienste werden auch als nachrichtenorientiere Middleware (*message oriented middleware*, MOM) bezeichnet.

Die direkte Kommunikation zur Ereignisübermittlung zwischen Server und Klienten bringt einige Nachteile mit sich. Dazu gehört die mögliche Unterbrechung der Kommunikation zwischen Klient und Server sowie der Ausfall eines der Systeme. Um diese Nachteile zu überwinden werden in der Regel persistente Warteschlangen für die Nachrichtenübermittlung verwendet. Der Server sendet ein Ereignis nicht direkt an einen Klienten, sondern fügt es in eine Warteschlange ein, aus welcher der Klient das Ereignis abrufen kann. Der Klient fragt in regelmäßigen Abständen bei der Warteschlange an, ob neue Nachrichten verfügbar sind.

Ein weiterer Vorteil dieser Methode gegenüber eines *Callback*-Verfahrens, bei dem der Klient dem Server eine *Callback*-Schnittstelle zur Verfügung stellen muss, ist, dass der Klient dem Server gegenüber nicht als Dienst auftreten muss.

Die Einfüge- bzw. Entnahme-Operationen von Warteschlangen sind meistens Teil einer Transaktion. So wird sichergestellt, dass ein Klient die Ereignisse, welche durch eine Transaktion erzeugt werden, auch in Ausnahmefällen korrekt empfängt oder dass diese bei Abbruch einer Transaktion nicht ausgeliefert werden.

2.3 Schnittstellen zu Informationssystemen

In diesem Abschnitt werden einige (Programmier-)Schnittstellen zu Datenquellen vorgestellt. Diese Schnittstellen werden auf ihre Tauglichkeit als Schnittstelle für die Integration von Informationssystemen hin untersucht. Dabei wird überprüft, in wie weit die Anforderungen, welche im vorhergehenden Abschnitt definiert worden sind, erfüllt werden.

2.3.1 JDBC

JDBC (*Java Database Connectivity*) [EH01] ist eine Programmierschnittstelle zum datenbank-unabhängigen Zugriff auf SQL-fähige relationale Datenbankmanagementsysteme (RDBMS). Um die Verbindung zu einer speziellen Datenbank-Software herzustellen ist ein Treiber notwendig, der einer dieser vier Kategorien angehört:

- **JDBC-ODBC Brücke mit ODBC Treiber**. Diese Treiber ermöglichen JDBC Zugang zu einer Datenbank über nativen ODBC [Gei95] Code, in der Regel Datenbank-Client-Code, der auf dem Rechner, der diesen Treiber benutzt, vorhanden sein muss.

- **Nativer Treiber, teilweise mit Java Technologie**. Diese Treiber konvertieren JDBC-Aufrufe in Aufrufe an die Client-Programmierschnittstelle einer Datenbank. Auch diese Treiber erfordern nativen Code auf jedem Rechner, der diesen Treiber benutzen soll.
- **Reiner Java Treiber für Datenbank Middleware**. Diese Treiber übersetzen JDBC-Aufrufe in das Protokoll einer Middleware, welche dann mit der eigentlichen Datenbank kommuniziert. Die Middleware bietet Zugang zu mehreren, unterschiedlichen Datenbanksystemen.
- **Reiner Java Treiber**. Ein Treiber dieser Kategorie übersetzt JDBC-Aufrufe direkt in das Netzwerk-Protokoll, welches von dem Datenbanksystem benutzt wird.

Die JDBC-API bietet Zugriff auf die Datenbank-Meta-Daten, damit eine Anwendung die speziellen Eigenschaften einer Datenbank ausnutzen kann. JDBC erlaubt es eine Verbindung zu einer Datenbank herzustellen, SQL-Ausdrücke an die Datenbank zu senden und die Ergebnisse von Anfragen zu verarbeiten.

Daten werden in relationalen Datenbanken in tabellarischer Form gespeichert. In Repositories werden Daten meistens in hierarchischen Strukturen verwaltet. Diese lassen sich zwar auf die Relationen in einer relationalen Datenbank abbilden, aber die Navigation innerhalb der Hierarchie wird dadurch erschwert. Abfragen eines Teilbaums sind mit SQL nicht leicht auszudrücken. Da JDBC für Zugriffe auf relationale Datenbanken entworfen wurde, eignet es sich nicht gut, um auf die Strukturen eines Repository-Systems zuzugreifen.

Da die JDBC-Schnittstelle sehr eng an SQL angelehnt ist, stellt sie auch dieselben Merkmale zur Verfügung, die eine SQL-fähige Datenbank besitzt. Dazu zählen in der Regel Datenzugriff und -manipulation, Transaktionen, Sperren, Anfragen und Volltextsuche. Zu den Merkmalen, die von SQL-Datenbanken und damit von JDBC im Allgemeinen nicht unterstützt werden, zählen Überwachung, Versionierung und Zugriffskontrolle.

Neuere Datenbankmanagementsysteme bieten zwar Überwachung in der Form von so genannten *Triggers* an. Diese erfordern aber einigen Programmieraufwand, um die Information über die Änderung an die Klienten der Datenbank zu übertragen. Es existiert somit keine einfache und einheitliche Möglichkeit um Änderungen an einer Datenbank zu verfolgen.

Zugriffkontrollmechanismen werden von allen Datenbanksystemen unterstützt. Es gibt jedoch noch keine einheitliche Möglichkeit, Zugriffsrechte für ein Datenbankobjekt, zum Beispiel eine Zeile in einer Tabelle, abzufragen.

Versionierung mit JDBC bzw. einer relationalen Datenbank zu lösen ist nicht einheitlich möglich. Dieses kann nur anwendungsspezifisch gelöst werden, indem das individuelle Datenbankschema Strukturen zur Versionierung von Datenbankobjekten vorsieht.

Aus den oben genannten Gründen eignet sich JDBC nicht besonders gut als einheitliche Schnittstelle für Repository-Systeme.

2.3.2 Java Data Objects

Java Data Objects (JDO) ist ein standardisiertes, schnittstellen-basiertes Persistenz-Abstraktionsmodell. Die *Java Data Objects*-Technologie kann dazu genutzt werden um

die Instanzen eines Java Domänenmodells direkt in einer Datenbank zu speichern. JDO benutzt dabei die Java-Klassen einer Anwendung als Datenmodell. Es ist nicht notwendig JDO-spezifische Typen zu benutzen um persistente Klassen zu definieren. In vielen Fällen können Java-Klassen in ihrer kompilierten Form in einer JDO-Umgebung genutzt werden. Um JDO-Persistenz zu Java-Klassen hinzuzufügen müssen diese „erweitert“ (*enhanced*) werden. Diese Erweiterung wird nach der Kompilierung in einem zusätzlichen Schritt von speziellen Werkzeugen ausgeführt. Die so erweiterten Klassen bieten transparenten Zugriff auf die Objekte in einer Datenbank.

JDO definiert Datendefinition und -manipulation, eine Abfragesprache und Transaktionen. Andere Dienste wie Versionierung oder Überwachung müssen anwendungsspezifisch realisiert werden.

Wie JDBC erfordert auch JDO einen Treiber oder eine spezielle Implementierung für jedes Datenbanksystem, auf das zugegriffen werden soll.

Der hier untersuchte Anwendungsfall der Integration von heterogenen Informationssystemen wird von JDO nur unzureichend unterstützt. JDO ist für den entgegengesetzten Fall entwickelt worden, um Java-Objekte in einem Datenspeicher transparent zu speichern. Um auf das Datenmodell einer bestimmten Quelle zuzugreifen müssen zur Benutzung von JDO zuerst Java-Klassen für die in dem Modell definierten Objekte angelegt werden.

Ein weiterer Nachteil von JDO besteht darin, dass es nicht möglich ist die Datenquelle zu erkunden, also zum Beispiel einfachen Zugriff auf das *Data Dictionary* zu erhalten. Die Objekte in der Datenbank müssen bekannt sein, um dafür Java-Klassen zu erstellen. Dieses erschwert eine Integration von Systemen, deren Aufbau nicht genau bekannt ist.

Da der hier untersuchte Anwendungsfall nicht ausreichend unterstützt wird, ist JDO nicht als Schnittstelle zum Zugriff auf Repository-Systeme geeignet.

2.3.3 Document Object Model

Das *Document Object Model* [HHW+03], kurz DOM, ist eine plattform-unabhängige und sprach-neutrale Schnittstelle, die es Programmen und Skripten erlaubt auf Dokumente zuzugreifen und deren Struktur, Inhalt und Stil zu ändern. DOM definiert eine Reihe von Objekten und Schnittstellen um auf Dokumentobjekte zuzugreifen und sie zu manipulieren. Die spezifizierte Funktionalität wird in der Regel auf HTML- und XML-Dokumente angewandt.

DOM repräsentiert Dokumente als eine Hierarchie von Knotenobjekten (Objekte vom Typ `Node`), die andere, speziellere Schnittstellen implementieren. Einige Knoten können verschiedene Kindknoten besitzen, andere sind Blätter, die keine weiteren Kinder in der Baumstruktur besitzen.

Das *Document Object Modell* wird fast ausschließlich zum Zugriff auf XML-Dokumente oder XML-ähnliche Strukturen, wie HTML, benutzt.

Als Schnittstelle zu Repository-Systemen eignet sich die DOM-Schnittstelle nicht, da keine höherwertigen Dienste, wie Transaktionen, Versionierung, Zugriffsschutz, Sperren und Überwachung definiert werden.

2.3.4 Java Content Repository

Die *Java Content Repository* Programmierschnittstelle [JCR03] wird speziell für einen universellen Zugriff auf Content Management Systeme und andere Arten von Repository-Systemen entwickelt. Alle zuvor genannten Anforderungen werden durch diese Schnittstelle abgebildet.

Zur Zeit wird diese Schnittstelle noch spezifiziert. Wenn diese Schnittstelle breite Herstellerunterstützung findet, ist davon auszugehen, dass sie sich als Standard-Schnittstelle für Repository-Systeme durchsetzen wird. Da der Experten-Gruppe viele Hersteller von Content und Datenbank Management Systemen angehören, ist mit einer schnellen Akzeptanz dieser Schnittstelle zu rechnen.

Da diese Schnittstelle alle Anforderungen gut erfüllt, wird sie hier für eine weitere Untersuchung ausgewählt. Außerdem wird sie als Schnittstelle zur Anbindung externer Repositories an die *CoreMedia Smart Content Technology* genutzt (siehe Abschnitt 5).

Die *Java Content Repository* Spezifikation wird in Abschnitt 3 genauer beschrieben.

3 Java Content Repository

Die *Content Repository API for Java Technology* (auch *Java Content Repository* oder kurz JCR) Spezifikation wird seit Anfang 2002 im *Java Community Process* entwickelt. Der Expertengruppe, die diese Spezifikation entwirft, gehören namhafte Firmen der IT-Branche an, wie Hewlett Packard, IBM, Bea Systems, Oracle, SAP, Sun Microsystems, Vignette und andere.

Übereinstimmungsebenen

Die *Java Content Repository* Spezifikation ist in zwei Übereinstimmungsebenen (*Compliance Levels*) unterteilt. Die erste Ebene legt grundlegende *Repository*-Funktionen fest. Die Zweite beschreibt weitergehende Funktionalität.

Eine Implementierung eines Repositories erfüllt die *Java Content Repository*-Spezifikation, wenn sie alle in einer Übereinstimmungsebene festgelegten Funktionen enthält.

Die Bestandteile der ersten Ebene sind:

- Abfragen und Durchlaufen von Knoten und Eigenschaften
- Lesen und Schreiben von Eigenschaftswerten
- Erzeugen und Löschen von Knoten und Eigenschaften
- Zuweisen von Knotentypen
- Suchen im Repository

Und Ebene 2 beinhaltet zusätzlich:

- *Hard Links*
- Transaktionen
- Versionierung
- Beobachtung
- Zugriffsberechtigungen
- Sperren

3.1 Datenmodell und Typsystem

Die Datenstruktur eines *Java Content Repositories* besteht aus einem Baum oder einem gerichteten azyklischen Graphen von Knoten und Eigenschaften. Durch Vater-Kind-Beziehungen zwischen Knoten bzw. zwischen Knoten und Eigenschaften wird eine Hierarchie definiert. Jeder Knoten kann beliebig viele Kindknoten und Kindeigenschaften besitzen.

Es gibt einen ausgezeichneten Knoten, den Wurzelknoten, der keinen Vaterknoten besitzt. Alle anderen Knoten haben mindestens einen Vaterknoten.

Jedem Knoten ist ein Knotentyp zugewiesen. Dieser muss bei der Erzeugung eines Knotens angegeben werden. Der Typ eines Knotens definiert, welche Kindknoten und Eigenschaften ein Knoten haben darf oder muss und wie diese Kindknoten und Eigenschaften beschaffen sein müssen (siehe auch Abbildung 7). Jeder Knotentyp in einem *Java Content Repository* besitzt die folgenden Attribute:

- **Name**
  Jeder in einem Repository bekannte Knotentyp muss einen eindeutigen Namen besitzen.
- **Supertypen**
  Ein Knotentyp kann einen anderen Knotentyp erweitern. Optional, wenn durch die Implementierung unterstützt, kann ein Knotentyp auch mehrere Knotentypen erweitern. Durch die Erweiterung erbt ein Knotentyp die Eigenschafts- und Kindknotendefinitionen des Supertypen. Mögliche Konflikte, wie zwei Supertypen, die gleich benannte Kindknoten- oder Eigenschaftsdefinitionen enthalten, werden dadurch vermieden, dass die Erzeugung eines gemeinsamen Subtypen nicht erlaubt ist.
- **Eigenschaftsdefinitionen**
  Eine Menge von `PropertyDef`-Objekten, die festlegt, welche Eigenschaften ein Knoten des definierten Typs haben darf und welche Charakteristika diese Eigenschaften aufweisen müssen. Zu diesen Charakteristika gehören:
  - **Name**
    Der Name der Eigenschaft, auf die diese Definition angewandt werden soll. Wird kein Name angegeben ist diese Definition eine so genannte „Restdefinition“ (*residual definition*). Diese Definition legt dann die Charakteristika für alle Eigenschaften fest, die nicht explizit benannt sind, d.h. es gibt keine Definition, deren Namen mit dem Namen der Eigenschaft übereinstimmt.
  - **Typ**
    Jede Eigenschaft muss genau einer der in Abbildung 8 gezeigten Typen besitzen.
  - **Wertbeschränkung**
    Mit der Wertbeschränkung können die möglichen Werte einer Eigenschaft begrenzt werden. Bei Eigenschaften vom Typ `PropertyType.LONG` können zum Beispiel der maximale und der minimale Wert festgelegt werden.
  - **Vorgegebener Wert**
    Der Wert, den die Eigenschaft besitzt, wenn sie ohne explizite Angabe eines Wertes erzeugt wird.
  - **Automatische Erzeugung**
    Gibt an, ob die definierte Eigenschaft automatisch erzeugt wird, wenn der Vaterknoten erzeugt wird.

  - **Verpflichtend**
    Gibt an, ob die Eigenschaft immer einen Wert besitzen muss. Das heißt, ob der `null` Wert zulässig ist oder nicht.
  - **Bei Versionierung**
    Definiert, was mit dieser Eigenschaft geschieht, wenn eine neue Version des Vaterknotens angelegt wird.
  - **Nur lesen**
    Durch dieses Attribut kann eine Eigenschaft ausschließlich für lesenden Zugriff markiert werden. Dadurch ist es nicht möglich die definierte Eigenschaft über die Programmierschnittstelle zu ändern.

- **Kindknotendefinitionen**
  Eine Menge von `ChildNodeDef`-Objekten, die spezifiziert, welche Kindknoten für Knoten des definierten Typs erlaubt sind und welche Charakteristika diese Kindknoten aufweisen müssen. Diese Charakteristika sind:

  - **Name**
    Der Name des Kindknotens, auf den diese Definition angewandt werden soll. Wird der Name ausgelassen, gilt diese Definition als Restdefinition.
  - **Knotentypbeschränkung**
    Die kleinste Menge von Knotentypen, die der definierte Kindknoten aufweisen muss.
  - **Vorgegebener Knotentyp**
    Wird bei der Erstellung eines Kindknotens kein Typ explizit angegeben, wird dem neuen Knoten dieser Typ zugewiesen.
  - **Automatische Erzeugung**
    Gibt an, ob der definierte Kindknoten automatisch erzeugt wird, wenn der Vaterknoten erzeugt wird.
  - **Bei Versionierung**
    Definiert, was mit diesem Kindknoten geschieht, wenn eine neue Version des Vaterknotens angelegt wird.
  - **Nur lesen**
    Durch dieses Attribut kann ein Kindknoten ausschließlich für lesenden Zugriff markiert werden. Dadurch ist es nicht möglich diesen Knoten über die Programmierschnittstelle zu ändern.

Ein Folgerung aus diesem, in Abbildung 7 gezeigten, Typ-Modell ist, dass die Charakteristika eines Knotens nicht ausschließlich durch dessen Typ festgelegt werden, sondern auch durch den Typ des Vaterknotens, dessen Kindknotendefinitionen die Charakteristika beschreibt.

Eigenschaften haben genau einen Vaterknoten und können keine Kinder besitzen. Sie sind die Blätter im Repository-Baum. In den Eigenschaften werden die eigentlichen Daten

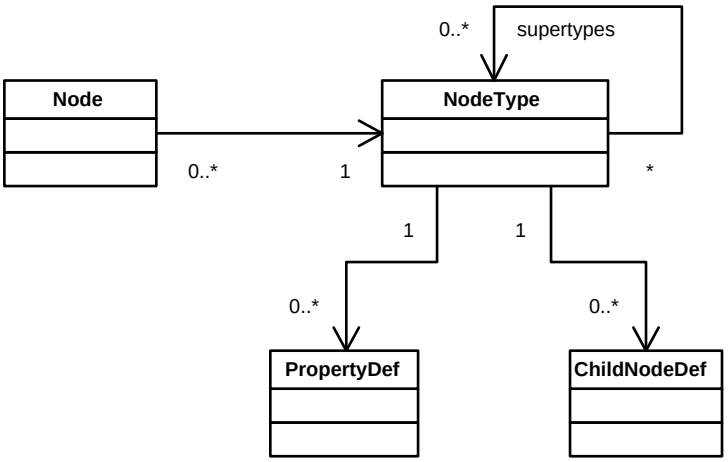

Abbildung 7: Typisierung im *Java Content Repository*

gespeichert. Eigenschaften sind Name-Wert-Paare und stellen die kleinste Einheit von Inhalten oder Informationen dar. Jede Eigenschaft besitzt genau einen der in Abbildung 8 aufgezählten Typen.

`PropertyType.STRING`
`PropertyType.BINARY`
`PropertyType.DATE`
`PropertyType.LONG`
`PropertyType.DOUBLE`
`PropertyType.BOOLEAN`
`PropertyType.SOFTLINK`

Abbildung 8: Die Eigenschaftstypen eines *Java Content Repository*s

In der Programmierschnittstelle werden Knoten durch Objekte vom Typ `javax.jcr.Node` repräsentiert und Eigenschaften durch Objekte mit der Schnittstelle `javax.jcr.Property`. Da Konten und Eigenschaften einige gemeinsame Funktionalitäten haben, wurde eine Schnittstelle `javax.jcr.Item` eingeführt, welche grundlegende Methoden enthält und durch `javax.jcr.Node` und `javax.jcr.Property` erweitert wird. Abbildung 9 zeigt diese Zusammenhänge zwischen den Schnittstellen.

Auf die Meta-Daten, auch Meta-Inhalte genannt, also Daten, welche die Inhalte in einem Repository beschreiben, kann ebenso über den Repository-Baum zugegriffen werden wie auf „richtige“ Inhalte. Dies ist deshalb möglich, weil die Meta-Daten ebenfalls als Knoten und Eigenschaften abgebildet werden. Auf die Meta-Daten eines Knoten, wie Typinformationen

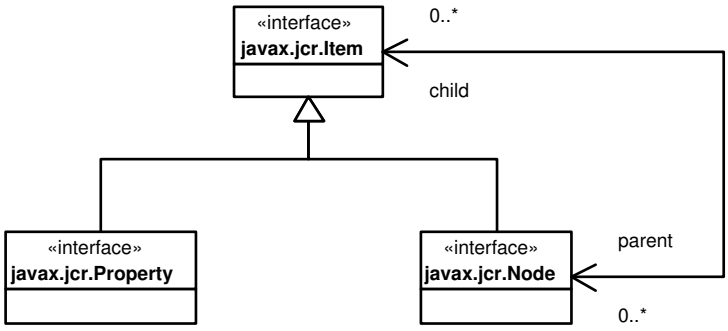

Abbildung 9: Das *Java Content Repository* Modell

oder die Versionshistorie, kann so über die gleiche Weise zugegriffen werden, wie auf die eigentlichen Inhalte. Dadurch ist es ebenfalls möglich diese Informationen über eine einheitliche Anfrageschnittstelle abzufragen. Die Meta-Informationen sind in Knoten mit vordefinierten Typen (siehe Abschnitt 3.1.2) gespeichert.

Für den Zugriff auf die im Repository enthaltenen Daten wird eine Programmierschnittstelle zur Verfügung gestellt, die unter anderem auf den bereits genannten Schnittstellen basiert.

3.1.1 Namen und Pfade

Jedes Element, also ein Knoten oder eine Eigenschaft, in einem *Java Content Repository* besitzen einen Namen. Daher kann ein Element in der Hierarchie durch seinen Pfad adressiert werden.

Durch den Einsatz von *Hard Links* (siehe Abschnitt 3.3.1) in Repositories der Übereinstimmungsebene 2 kann ein Element durch mehrere Pfade identifiziert werden.

Für die Zeichenkettenrepräsentation eines Pfades wird eine Notation verwendet, wie von Dateisystemen bekannt. Zum Beispiel identifiziert der Pfad `/a/b/c` das Element `c` mit dem Vaterknoten `b` und dem Großvaterknoten `a`. Der Knoten `a` ist ein Kind des Wurzelknotens.

Der Wurzelknoten wird durch den Pfad `/` adressiert.

3.1.2 Vordefinierte Knotentypen

Jedes *Java Content Repository* muss eine fest vorgeschriebene Menge von vordefinierten Knotentypen unterstützen. Die Namen dieser Knotentypen besitzen das Präfix `nt`, um sie von Benutzer-definierten Typen zu unterscheiden.

Im Folgenden wird auf einige der vordefinierten Knotentypen genauer eingegangen, da sie für einige Funktionen des *Java Content Repositories* wichtig sind. Repository-Merkmale wie Versionierung und *Hard Links* kommen nicht ohne die eingebauten Knotentypen aus. Die Beziehungen zwischen den verschiedenen Knotentypen werden in Abbildung 10 dargestellt.

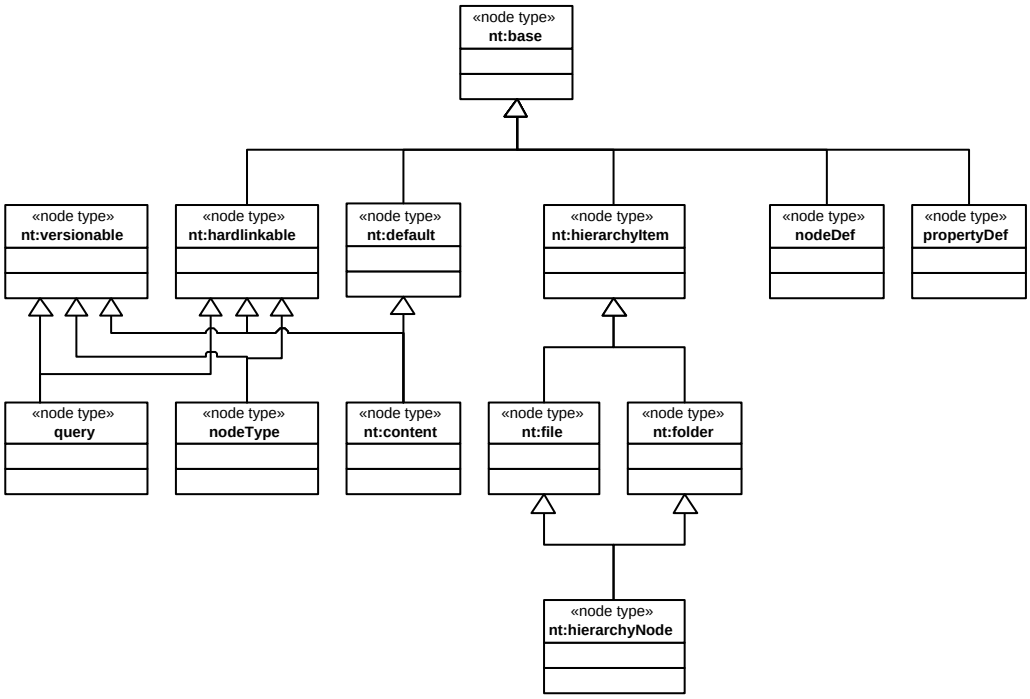

Abbildung 10: Die vordefinierten Knotentypen im *Java Content Repository*

nt:base

Alle Knotentypen erben von `nt:base`. Dieser Knotentyp definiert eine spezielle Eigenschaft `jcr:nodeType`, die jeder Knoten haben muss. Diese Eigenschaft vom Typ `PropertyType.STRING` enthält den Typ des Knotens als Wert. So kann auf den Typ eines Knoten leicht über eine Eigenschaft zugegriffen werden. Da die Knotentypinformation in den Inhalten selbst gespeichert ist, kann sie somit auch neben allen anderen Inhalten serialisiert und deserialisiert werden (siehe Abschnitt 3.2.2). Auf die `jcr:nodeType` Eigenschaft kann nur lesend zugegriffen werden. Sie wird von der Repository-Implementierung verwaltet und kann nicht über die Programmierschnittstelle geändert werden.

nt:hardlinkable

Damit ein Knoten Ziel eines *Hard Links* sein kann, muss er vom Typ `nt:hardlinkable` sein. Dieser Typ definiert eine automatisch erzeugte, verpflichtende Eigenschaft mit dem Namen `jcr:uuid` vom Typ `PropertyType.STRING`. Der Wert dieser Eigenschaft wird von der Repository-Implementierung bei der Erzeugung des Vaterknotens gesetzt. Die Eigenschaft kann nach der Erstellung nicht geändert werden. Eine genauere Beschreibung von *Hard Links* befindet sich in Abschnitt 3.3.1.

nt:hierarchyItem, nt:file, nt:folder, nt:hierarchyNode

Die *Java Content Repository* Spezifikation sieht keine strikte Trennung von Hierarchie und Inhalten vor. Insbesondere gibt es kein klares Konzept eines Ordners oder Verzeichnisses. Um trotzdem eine gewisse Trennung vorzusehen, wurden die Knotentypen `nt:hierarchyItem`, `nt:file`, `nt:folder` und `nt:hierarchyNode` eingeführt. Durch direkte Benutzung dieser Typen oder durch Erstellung von Subtypen davon, lässt sich auf Meta-Datenebene eine Trennung von Hierarchie und Inhalten durchführen. Die Benutzung dieser Typen wird jedoch nicht erzwungen und ist letztendlich anwendungsspezifisch.

nt:versionable

Dieser Knotentyp wird für die Versionsverwaltung (siehe Abschnitt 3.3.3) benötigt. Damit ein Knoten versionierbar ist, muss er von diesem Typ sein. Auf alle Eigenschaften die durch diesen Knotentyp definiert werden, kann nur lesend zugegriffen werden. Sie werden nur von der Repository-Implementierung erzeugt und geändert.

nt:propertyDef, nt:nodeDef, nt:nodeType

Knoten dieser Typen werden benutzt um die Definitionen von Knotentypen zu speichern (siehe auch Seite 23).

nt:query

Knoten vom Typ `nt:query` speichern eine Anfrage (siehe Abschnitt 3.2.3). Die definierten Eigenschaften können einen Anfrageausdruck in einer von der Repository-Implementierung unterstützten Anfragesprache enthalten. Es wird also nicht ein Anfrageergebnis aufgezeichnet, sondern die Definition einer Anfrage, um sie zu einem späteren Zeitpunkt erneut ausführen zu können.

3.1.3 Isolierung

Jedes *Java Content Repository* ist in drei Isolationsschichten aufgebaut. Diese Schichten sollen die Arbeit in Mehrbenutzerumgebungen unterstützen und die Benutzer voneinander isolieren.

- **Repository-Schicht**. Dieses ist die Persistenzschicht, welche die Inhalte permanent speichert. Sie wird durch ein Objekt vom Typ `javax.jcr.Repository` repräsentiert.
- **Workspace-Schicht**. Ein Repository kann einen oder mehrere *Workspaces* unterstützen. Ein *Workspace* isoliert dessen Benutzer von Benutzern anderer *Workspaces*. Änderungen an einem Workspace sind zunächst nur für dessen Benutzer sichtbar. Die Änderungen an einem *Workspace* müssen durch eine *Check-In*-Operation explizit an die Repository-Schicht übertragen werden, um dort persistent gespeichert zu werden. Damit werden die Änderungen für alle Benutzer des Repositorys sichtbar.

- **Transiente Schicht**. Wenn ein Knoten oder eine Eigenschaft aus einem *Workspace* abgefragt werden, stellen die dadurch erzeugten transienten Java-Objekte die höchste Isolierungsschicht dar. Änderungen an diesen Objekten sind zunächst nur für den Benutzer sichtbar, der diese Änderungen durchführt. Diese Änderungen können in die *Workspace*-Schicht gespeichert werden, um sie für dessen Benutzer sichtbar zu machen.

3.2 Grundlegende Funktionen

In diesem Abschnitt werden die grundlegenden Funktionen eine *Java Content Repository*s nach Übereinstimmungsebene 1 beschrieben. Zu diesen Funktionen zählen allgemeiner Datenzugriff, XML-Serialisierung, Knotentypen und das Suchen im Repository.

3.2.1 Datenzugriff

Lesender Zugriff auf das Repository beinhaltet den Zugriff auf Knoten, entweder direkt oder durch Traversieren des Repository-Graphen, und das Auslesen von Eigenschaftwerten. Die Funktionalität wird durch Methoden in den Schnittstellen `javax.jcr.Item`, `javax.jcr.Property` und `javax.jcr.Node` bereitgestellt.

Um Daten in das Repository zu schreiben, müssen zunächst die Knoten oder Eigenschaften gefunden werden, an welchen die Änderungen vorgenommen werden sollen. Dann können neue Knoten oder Eigenschaften eingefügt, die Werte bestehender Eigenschaften geändert oder Knoten und Eigenschaften gelöscht werden.

Wird ein Wert für eine Eigenschaft gesetzt und dieser entspricht nicht dem Typ der Eigenschaft, so wird eine Konvertierung gemäß den Regeln für Typkonvertierungen aus der *Java Content Repository* Spezifikation versucht. Dabei können alle Typen in Zeitenketten konvertiert werden (bei `PropertyType.BINARY` jedoch nur eingeschränkt), bzw. können Zeichenketten in jeden Typ konvertiert werden, sofern sie einem festgelegten Format unterliegen.

Diese Änderungen existieren zunächst nur in der transienten Schicht (siehe Abschnitt 3.1.3). Um die Änderungen in die *Workspace*-Schicht zu übertragen, muss der Client diese Änderungen speichern. Dieses geschieht über die Methode `javax.jcr.Node.save`, welche die Änderungen an dem Unterbaum, auf dessen Wurzelknoten diese Methode aufgerufen wurde, in *Workspace*-Schicht überträgt.

Sind die Änderungen in der *Workspace*-Schicht gespeichert, können sie über die Methode `javax.jcr.Node.checkin` in die Repository-Schicht übertragen werden. Bei einem Repository der Übereinstimmungsebene 2 wird dabei auch gleichzeitig eine neue Version des Knotens angelegt.

Änderungen, die in der transienten Schicht vorgenommen werden, können beliebig inkonsistent sein. Das heißt, dass die Kindknoten und -eigenschaften eines Knotens nicht unbedingt der Typdefinition des Knotens entsprechen müssen. Diese Aufweichung des Typsystems ist notwendig, um komplexere Änderungsoperationen, die nur mit den elementaren Operationen, welche die Schnittstellen `javax.jcr.Item`, `javax.jcr.Property` und `javax.jcr.Node` zu Verfügung stellen, möglich zu machen. Die Einschränkungen, die

durch den Typ eines Knotens gefordert werden, werden aber beim Speichern in die *Workspace*-Schicht geprüft und eingefordert.

3.2.2 XML-Serialisierung

Die *Java Content Repository* Spezifikation definiert zwei Abbildungen des Repository-Datenmodells nach XML [BPSMM00]. Diese Abbildungen werden „System-Sicht“ und „Dokument-Sicht“ genannt. Die System-Sicht enthält alle verfügbaren Informationen, ist aber daher für einen Menschen nicht gut lesbar. Die Dokument-Sicht enthält nur die notwendigen Informationen und ist dadurch wesentlich besser lesbar. Ebenso lassen sich für die Dokument-Sicht einfachere XPath-Anfragen [CD99] stellen.

XML Schemata [Fal01, TBMM01, BM01] für die beiden Abbildungen sind in der aktuellen Version der *Java Content Repository* Spezifikation noch nicht enthalten, sollen aber bis zur endgültigen Version vorliegen.

Die System-Sicht basiert auf folgenden Konventionen:

- Jeder JCR-Knoten wird auf ein XML-Element mit dem Namen `sv:node` abgebildet.
- Jede JCR-Eigenschaft wird auf ein XML-Element mit dem Namen `sv:property` abgebildet.
- Der Name jedes Knoten oder jeder Eigenschaft wird in einem `sv:name` Attribut in dem zugehörigen XML-Element abgelegt.
- Der Eigenschaftstyp wird in einem XML-Attribut mit dem Namen `sv:type` gespeichert.
- Der Wert jeder JCR-Eigenschaft wird als Textknoten in dem zugehörigen XML-Element gespeichert.
- Die Hierarchie der JCR-Knoten und -Eigenschaften wird durch die Hierarchie der entsprechenden XML-Elemente abgebildet.

Die Dokument-Sicht basiert auf diesen Konventionen:

- Jeder JCR-Knoten wird auf ein XML-Element mit dem Namen des Knoten abgebildet.
- Jede JCR-Eigenschaft wird auf ein XML-Attribut mit dem Namen der Eigenschaft abgebildet.
- Der Wert jeder JCR-Eigenschaft wird zu dem Wert des zugehörigen XML-Attributs.
- Die Hierarchie der JCR-Knoten und -Eigenschaften wird durch die Hierarchie der entsprechenden XML-Elemente abgebildet.

Für diese XML-Serialisierungen gibt es zwei Anwendungsfälle:

- **Unterstützung von XPath Anfragen**. Die *Java Content Repository* Spezifikation definiert Anfragen, welche durch die XPath Anfragesprache ausgedrückt werden können. Da XPath entworfen wurde um Anfragen an einen XML-Baum zu stellen, ist es notwendig, eine XML-Abbildung für den Repository-Graphen zu finden, an die eine XPath-Anfrage gestellt werden kann. Dieses erlaubt die volle Semantik jeder XPath-Anfrage innerhalb JCR-Umgebung. Beide XML-Abbildungen können benutzt werden, um XPath-Anfragen zu definieren. Die Dokument-Sicht ist aber in der Regel die komfortablere.
- **Daten Im- und Export**. Die XML-Abbildungen können genutzt werden, um Repository-Inhalte in einer serialisierten Form zu exportierten bzw. zu importieren.

3.2.3 Anfrageschnittstelle

Um komplexe Anfragen an das Repository zu stellen, definiert die JCR-Spezifikation eine Anfrageschnittstelle. Diese Schnittstelle ermöglicht es Anfragen in beliebigen Anfragesprachen zu stellen, sofern diese von der Implementierung unterstützt werden. Der aktuelle Stand der Spezifikation schreibt jedoch vor, dass jede Implementierung XPath- und JCRQL-Anfragen unterstützen muss.

Da die logische Struktur der Inhalte in einem *Java Content Repository* nicht exakt der Struktur eines XML-Dokuments gleicht, muss die Repository-Struktur in eine XML-Struktur abgebildet werden, um XPath-Anfragen an ein Repository stellen zu können. Diese Abbildung geschieht auf einer der in Abschnitt 3.2.2 beschriebenen Arten.

Die JCRQL (*Java Content Repository Query Language*) ist eine SQL-ähnliche Anfragesprache. Eine formale Definition der Sprache befindet sich in Anhang A. Der Ausgangspunkt für die Syntaxbeschreibung ist das nicht-terminal Symbol `query`. Ein JCRQL-Ausdruck besteht immer aus einer `SELECT`-Klausel und optionalen `FROM`-, `LOCATION`-, `WHERE`-, `TEXTSEARCH`- und `ORDER BY`-Klauseln.

Mit Hilfe der `SELECT`-Klausel können Knoten anhand ihres Namens ausgewählt werden. Alternativ kann auch das Platzhalterzeichen * verwendet werden, um alle Knoten auszuwählen.

Die `FROM`-Klausel erlaubt es, die Anfrage so einzuschränken, dass ihr Ergebnis nur Knoten bestimmter Typen enthält. Auch hier ist wieder das *-Zeichen erlaubt, um alle Knotentypen auszuwählen.

Eine Einschränkung des Anfrageergebnisses auf einem bestimmten Bereich in der Repository-Hierarchie ist durch die `LOCATION`-Klausel möglich. Die Einschränkung wird durch eine Angabe eines Musters für Repository-Pfade realisiert. Das Ergebnis der Anfrage enthält dann nur Knoten deren Pfade diesem Muster entsprechen. Mit der optionalen `DEPTH`-Unterklausel ist es weiterhin möglich die Suchtiefe im Repository-Graphen einzuschränken. Die `FOLLOWING`-Unterklausel ändert den Gültigkeitsbereich des Musters für Repository-Pfade in der `LOCATION`-Klausel. Wird die `FOLLOWING`-Klausel ausgelassen, bezieht sich der „alle Nachfahren“-Operator (`//`) auf Nachfahren eines Knoten im Sinne von Eigenschaften und

Kindknoten. Wird jedoch die FOLLOWING-Klausel eingesetzt, bezieht sich dieser Operator auf Eigenschaften und Knoten, auf die durch Eigenschaften vom Typ PropertyType.SOFTLINK verwiesen wird. Wird zusätzlich noch das Schlüsselwort CHILDREN eingefügt, werden auch die eigentlichen Eigenschaften und Kindknoten durchsucht.

Die WHERE-Klausel erlaubt es, die Menge der durch die Anfrage gelieferten Knoten einzuschränken, indem Werte oder Bereiche von Werten für die Werte von Eigenschaften dieser Knoten vorgegeben werden.

Eine Sortierung des Anfrageergebnisses ist mit der ORDER BY-Klausel möglich.

Die TEXTSEARCH-Klausel ermöglicht die Einbettung eines Ausdrucks einer Volltextsuchsprache. Alle Repository-Implementierung müssen mindestens die einfache Suchmaschinensprache unterstützen, wie durch das simplesearch nicht-terminal Symbol in der JCRQL Definition angegeben. Diese Syntax basiert auf der Syntax von WWW-Suchmaschinen wie Google. Die Semantik dieser einfachen Ausdrücke ist:

- Terme, die durch Leerzeichen getrennt sind, werden implizit durch ein logischen UND verbunden.
- Terme können durch die explizite Angabe des Schlüsselwortes OR durch ein logisches ODER verbunden werden.
- Terme können durch Voranstellung des Minuszeichens (-) ausgeschlossen werden. Das heißt, das Ergebnis enthält nicht die ausgeschlossenen Terme.
- Ein Term ist ein einzelnes Wort oder ein Ausdruck, der von einfachen Anführungszeichen (') umschlossen ist.
- Der gesamte Suchausdruck kann in doppelte Anführungszeichen (") eingeschlossen werden, damit er Leerzeichen enthalten kann.
- Innerhalb des Suchausdrucks muss allen Literalen, die einfache oder doppelte Anführungszeichen und Minuszeichen sind, ein umgekehrter Schrägstrich (\) vorangestellt werden. Ein umgekehrter Schrägstrich muss dadurch auch einen umgekehrten Schrägstrich vorangestellt bekommen.

Der Gültigkeitsbereich der TEXTSEARCH-Klausel ist der Durchschnitt dieser zwei Mengen:

- Die Werte der Eigenschaften, die direkte Kinder der durch die anderen Klauseln spezifizierten Knoten sind.
- Der Inhalt der Volltextindices des Repositories. Der Inhalt dieser Indices ist implementierungs-spezifisch und wird nicht durch die *Java Content Repository* Spezifikation vorgegeben.

Auf das Ergebnis einer Anfrage kann sequentiell zugegriffen werden. Dabei ist es möglich über die Knoten in der Reihenfolge zu iterieren, wie sie in der ORDER BY-Klausel angegeben wurde, oder nach der Relevanz, die von der Volltextsuchmaschine ermittelt wurde, falls

die TEXTSEARCH-Klausel angegeben wurde. Welche Metrik zur Bestimmung der Relevanz benutzt wird, ist dabei von der Implementierung der Volltextsuchmaschine abhängig und nicht vorgeschrieben.

3.3 Erweiterte Funktionen

Dieser Abschnitt gibt einen Überblick über Repository-Anforderungen, die eine Implementierung erfüllen muss, um in die Übereinstimmungsebene 2 eingestuft zu werden.

3.3.1 Harte Verweise

In einem Repository der Ebene 2 kann ein Knoten mehrere Vaterknoten besitzen. Wird einem Knoten ein weiterer Vaterknoten zugewiesen, so wird diese Verknüpfung „harter Verweis“ (*hard link*) genannt. Diese Verknüpfung ist eine gerichtete Verknüpfung von dem neuen Vaterknoten zu dem verknüpften Knoten. Der zu verknüpfende Knoten wird auch Zielknoten der Verknüpfung genannt, wohingegen der neue Vaterknoten der Ursprung der Verknüpfung ist.

Wird ein Knoten gelöscht, so werden dessen Kindknoten nur genau dann gelöscht, wenn diese keine weiteren Vaterknoten besitzen. Dieses ist ein Unterschied zu einem Repository der Ebene 1, wo bei Löschung eines Knotens immer alle seine Kindknoten gelöscht werden.

Die Beziehung zwischen dem Knoten und dem ersten Vaterknoten unterscheidet sich nicht zu den Beziehungen mit weiteren Vaterknoten. Es gibt also keinen ausgezeichneten Vaterknoten, als dessen Kind ein Knoten ursprünglich erzeugt wurde. Dieses erscheint sinnvoll, da es denkbar ist, dass dieser erste Vaterknoten gelöscht wird, wobei ein durch einen anderen harten Verweis gebundener Kindknoten nicht mitgelöscht wird.

Um einem Knoten einem weiteren Vaterknoten zuordnen zu können, ihn also zum Ziel eines harten Verweises machen zu können, muss dieser Knoten vom Typ `nt:hardlinkable` sein. Wird ein Knoten von diesem Typ erzeugt, muss die Implementierung den Wert der `jcr:uuid` Eigenschaft auf einen allgemein eindeutigen Wert setzen. Durch diese UUID (*Universally Unique IDentifier*, allgemein eindeutiger Bezeichner) kann auf einen Knoten auch direkt zugegriffen werden, ohne das durch die Repository-Hierarchie navigiert werden muss. Für eine weitere Diskussion über die Beschaffenheit einer UUID siehe [DCE97, Anhang A]. Das Format einer UUID wird nicht von der *Java Content Repository* Spezifikation festgelegt und hängt damit von der jeweiligen Implementierung ab.

Eine weitere Einschränkung bei der Erzeugung von *Hard Links* ist die, dass durch einen harten Verweis keine Zyklen im Graphen der Repository-Hierarchie entstehen dürfen. Ein Zyklus entsteht, wenn ein Knoten in der Menge seiner Nachfahren enthalten ist. Der durch harte Verweise entstehende Graph muss ein azyklischer, gerichteter Graph sein.

Bei der Erstellung eines harten Verweises kann für den Zielknoten ein neuer Name angegeben werden. Unter diesem Namen ist der Knoten dann als Kindknoten des neuen Vaterknotens bekannt. In bestehenden Beziehungen zu anderen Vaterknoten bleibt der Knoten aber unter seinen bisherigen Namen bekannt. Zum Beispiel kann ein Knoten mit dem Pfad `/a/b` bei Verknüpfung mit dem Knoten `/c` einen neuen Namen `d` erhalten. Dieser Knoten

ist dann durch die Pfade `/a/b` und `/c/d` erreichbar. Dieses impliziert, dass der Name eines Knotens nicht Teil dieses Knotens ist sondern vielmehr Teil des Vaterknotens.

Eine mögliche Anwendung von *Hard Links* ist die Unterstützung von mehreren orthogonalen Hierarchien, die eine gemeinsame Menge von Inhalten referenzieren.

3.3.2 Transaktionen

Die Übereinstimmungsebene 2 der *Java Content Repository* Spezifikation fordert, dass jede Repository-Operation an eine Transaktion gebunden ist. Die Programmierschnittstelle bietet dazu die Möglichkeit, ein Repository an jede Standard-XA-Transaktion [XA92] zu binden.

Die Spezifikation erfordert weiterhin, dass eine Implementierung jede Repository-Operation implizit in einer eigenen Transaktion ablaufen lässt, wenn durch den Programmierer keine Transaktionsklammern explizit gesetzt wurden.

Jedes Repository kann als Ressource in einem JTA-kompatiblen J2EE Transaktionsmonitor genutzt werden [JTA99, Sha01].

3.3.3 Versionierung

Das Versionskontrollsystem ist in der vorliegenden Fassung 0.9.5 der *Java Content Repository* Spezifikation [JCR03] noch nicht endgültig spezifiziert. Es bestehen aber schon Ansätze, die in diesem Absatz kurz vorgestellt werden.

Als Vorlage für die Versionierung von Knoten und Eigenschaften soll, soweit wie möglich, der *Java Specification Request 147 (Workspace Versioning and Configuration Management)* [WVC03] benutzt werden. Diese Vorlage basiert jedoch auf einem Verzeichnis/Dokument-Paradigma, wohingegen ein *Java Content Repository* auf einem feinkörnigeren Knoten/Eigenschaften-Paradigma aufbaut.

Erweiterte Versionierungsmerkmale wie 3-Wege-Mischen (*3-way-merge*) mit einer gemeinsamen Vorgängerversion, Grundlinien und Konfigurationen sollen jedoch nicht in die *Java Content Repository* Spezifikation einfließen.

In einem Repository der Übereinstimmungsebene 2 können Knoten unter Versionskontrolle gestellt werden. Der Zustand eines solchen Knoten kann zu jedem beliebigen Zeitpunkt festgehalten und aufgezeichnet werden, sodass der Zustand des Knotens zu einem späteren Zeitpunkt wieder abgerufen werden kann.

Ein Knoten, der unter Versionskontrolle gestellt werden kann, wird versionierbarer Knoten genannt. Nicht alle Knoten in einem Repository sind versionierbar. Damit ein Knoten versionierbar ist, muss er mindestens vom Typ `nt:versionable` sein.

Steht ein Knoten unter Versionskontrolle, wird er versionierter Knoten genannt. Ein Knoten muss explizit unter Versionskontrolle gestellt werden. Direkt nach der Erzeugung ist ein Knoten nicht versioniert. Wird eine neue Version n' eines Knotens n angelegt, so heißt n' Nachfolgeversion von n und n ist die Vorgängerversion von n'. Wird ein Knoten unter Versionskontrolle gestellt, wird eine Versionshistorie für ihn angelegt. Die Versionshistorie beinhaltet den Versionsgraphen. Dieser Graph enthält alle Versionen eines Knotens. Die Struktur dieses Graphen sieht wie folgt aus:

- Der Versionsgraph besteht aus einer oder mehreren Versionen eines Knotens.
- Der Graph hat exakt eine Version, welche die Wurzel des Graphen bildet.
- Die Wurzelversion hat keine Vorgängerversion.
- Alle anderen Versionen (außer der Wurzelversion) können eine oder mehrere Vorgängerversionen haben.
- Jede Version kann eine oder mehrere Nachfolgerversionen haben.
- Eine Version kann nicht in der Menge ihrer Nachfolger enthalten sein. So entsteht ein gerichteter, azyklischer Graph.

Ein versionierter Knoten kann ausgeliehen oder zurückgegeben werden. Wenn ein Knoten zurückgegeben ist, können seine Eigenschaften nicht verändert werden und es können keine Kindknoten hinzugefügt oder entfernt werden. Direkt nachdem ein Knoten unter Versionskontrolle gestellt wurde befindet er sich in dem zurückgegebenen Zustand.

Ein zurückgegebener Knoten kann ausgeliehen werden, um seine Eigenschaften zu ändern bzw. um Kindknoten hinzuzufügen und zu entfernen. Wenn ein ausgeliehener Knoten zurückgegeben wird, wird eine neue Version dieses Knotens angelegt und diese wird als Nachfolgeversion der aktuellen Version des Knotens in die Versionshistorie eingetragen.

3.3.4 Überwachung

Ein Repository der Ebene 2 kann ein Benachrichtigungsmodell zur Verfügung stellen, dass einer Anwendung ermöglicht Änderungen an Repository-Inhalten zu verfolgen und gegebenenfalls Einspruch gegen diese Änderungen einzulegen.

Dazu kann eine Implementierung Benachrichtigungen über Ereignisse (*events*) an Objekte vom Typ `EventListener` bzw. `VetoableEventListener` verschicken. `EventListener`-Objekte werden asynchron benachrichtigt und sehen alle Ereignisse erst nachdem sie aufgetreten sind und die Transaktion abgeschlossen wurde. `VetoableEventListener`-Objekte werden synchron benachrichtigt. Sie sehen die Ereignisse bevor die Implementierung die zugehörige Aktion ausführt. Dadurch ist es möglich ein Veto gegen diese Aktion einzulegen. Dieses Veto führt dazu, dass eine Transaktion rückgängig gemacht wird.

Es können von der Repository-Implementierung Benachrichtigungen über die folgenden Ereignisse verschickt werden:

- Ein Element (ein Knoten oder eine Eigenschaft) wurde zum Repository hinzugefügt.
- Ein Element wurde geändert.
- Ein Element wurde gelöscht.
- Es wurde eine neue Version eines Elements angelegt.
- Es wurde eine Versionskennzeichnung (*label*) hinzugefügt.

- Es wurde eine Versionskennzeichnung entfernt.
- Ein Element wurde gesperrt.
- Ein Element wurde entsperrt.
- Eine Sperre ist abgelaufen.

3.3.5 Zugriffskontrolle

Die *Java Content Repository* Spezifikation beschreibt lediglich Methoden innerhalb der Programmierschnittstelle, die es erlauben festzustellen, welche Zugriffsrechte ein Benutzer auf einem Knoten oder eine Eigenschaft besitzt. Ein Mechanismus um Zugriffsrechte zu erteilen bzw. zu verweigern wird nicht beschrieben und muss auf Implementierungsebene gelöst werden.

Alle Methoden zum Zugriff auf Repository-Elemente müssen die Zugriffsrechte berücksichtigen. Findet ein nicht berechtigter Zugriff auf ein Element statt, wird eine Ausnahme ausgelöst. Eine Methode, die einen Iterator zurückliefert, muss Elemente, auf die ein Benutzer keinen Zugriff hat, aus dem Iterator ausschließen, sie also ignorieren.

3.3.6 Sperren

In diesem Abschnitt wird die Semantik von Sperren in einem *Java Content Repository* beschrieben.

Eine Sperre wird von einem Benutzter auf einem Knoten oder einer Eigenschaft gesetzt. Dieses Element wird dann als gesperrt bezeichnet. Der Benutzer, der diese Sperre gesetzt hat, wird als Besitzer der Sperre bezeichnet. Ob eine Sperre auf Knoten und Eigenschaften gesetzt werden oder nur auf Knoten und damit implizit auf Eigenschaften gesetzt wird, wird durch den Grad der Sperre definiert, den die konkrete Implementierung unterstützt.

Auf ein gesperrtes Element kann nur vom Besitzer der Sperre voll zugegriffen werden. Alle anderen Benutzter haben nur eingeschränkten Zugriff auf dieses Element. Welche Einschränkungen genau gelten, hängt vom Typ der Sperre ab. Eine Schreibsperre verhindert zum Beispiel, dass alle anderen Benutzer (außer dem Besitzer der Sperre) das betreffende Element ändern können. Die Typen, die eine Sperre besitzen kann, hängen von der jeweiligen Implementierung des Repositorys ab.

Eine Sperre kann entweder exklusiv (*exclusive lock*) oder nicht exklusiv (*shared lock*) benutzt werden. Dieses wird der Gültigkeitsbereich der Sperre genannt. Eine Implementierung kann Sperren eines oder beider Gültigkeitsbereiche unterstützen. Eine exklusive Sperre verhindert, dass andere Sperren auf ein Element gesetzt werden können. Eine nicht exklusive Sperre erlaubt es andere, nicht exklusive Sperren auf einem Element zu setzten. Es können jedoch keine exklusiven Sperren auf ein nicht exklusiv gesperrtes Element gesetzt werden. Nicht exklusive Sperren erlauben es mehr als einem Besitzer Zugriff auf ein Element zu haben, während anderen Benutzern kein Zugriff gewährt wird.

3.4 Produkte und Projekte

Im Folgenden werden zwei Produkte und Projekte vorgestellt, die auf der *Java Content Repository* Spezifikation basieren oder die dort spezifizierte Programmierschnittstelle nutzen.

3.4.1 Das Jakarta Slide Projekt

Das Jakarta Slide Projekt ist ein Unterprojekt des Jakarta Projekts der Apache Software Foundation. Das Jakarta Projekt erstellt und wartet Software-Lösungen basierend auf der Java Plattform. Diese Lösungen sind frei verfügbar und es müssen keine Lizenzkosten dafür gezahlt werden.

Der Hauptteil des Slide Projekts besteht aus einem Modul für Content Management und Integration. Dieses Modul stellt ein Rahmenwerk für Content Management auf einer niedrigen Stufe dar. Konzeptuell ermöglicht es die Verwaltung von binären Inhalten in einer hierarchisch organisierten Struktur, die wiederum in beliebigen Datenspeichern gehalten werden kann. Zusätzlich bietet das Slide Projekt Dienste wie Versionierung, Sicherheit und Indizierung.

Slide kann Daten aus externen Datenquellen integrieren und verwalten. Für jede Datenquelle muss eine schmale Abstraktionsschicht entwickelt werden. Dadurch kann Slide Daten aus verschiedenen physikalischen Orten auf eine einheitliche, hierarchische Weise integrieren. Die Anwendungsmöglichkeiten von Slide reichen von der Verwaltung von Intranet-Anwendungsinhalten bis zu Datei-Servern.

Zum Datenzugriff bietet Slide ein WebDAV-Modul. WebDAV [Whi98, WW98] ist ein Standard der IETF, der von Firmen, wie Microsoft, IBM, Novell und Adobe, unterstützt wird und erweitert HTTP [FGM+99] um die nötigen Methoden für verteiltes Web-Authoring. Auf alle Daten, die von Slide verwaltet werden, kann über WebDAV zugegriffen werden. Dadurch ist eine verteilte Verwaltung und Manipulation der von Slide verwalteten Daten mit Hilfe von Standard-Werkzeugen von Drittanbietern möglich.

Die Referenzimplementierung, die nach der Definition des *Java Community Processes* für eine Anerkennung der *Java Content Repository* Spezifikation notwendig ist, wird im Rahmen des Slide Projektes entwickelt. Zum jetzigen Zeitpunkt bietet diese Implementierung ein *Java Content Repository* der Übereinstimmungsebene 1 (siehe Abschnitt 3), nach Aussagen des zuständigen Entwicklers soll jedoch bis zur Verabschiedung der Spezifikation eine vollständige Ebene-2-Implementierung vorliegen. Als Datenspeicher unterstützt die Referenzimplementierung ein lokales Dateisystem oder ein verteiltes Dateisystem, auf welches über das WebDAV-Protokoll zugegriffen wird.

3.4.2 Obinary Magnolia

Magnolia ist eine freies, J2EE-kompatibles Content Management System, welches mittlerweile in der Version 1.1 vorliegt. Es wird als Open Source Projekt von der Firma Obinary verwaltet und entwickelt. Magnolia basiert vollständig auf der Java 2 Plattform und benutzt die in der *Java Content Repository* Spezifikation definierte Programmierschnittstelle zum Zugriff auf Inhalte. Zur Verwaltung der Inhalte steht eine Web-Benutzeroberfläche zur Verfügung. Es

gibt eine Programmierschnittstelle und *Tag*-Bibliotheken zum Entwickeln von Vorlagen mit *Java Servlets* [Cow01] und *Java Server Pages* [RPL03].

Magnolia ist das erste Open Source Content Management System, welches den *Java Content Repository* Standard unterstützt.

Die wichtigsten Merkmale von Magnolia sind:

- **100% Java und J2EE kompatibel**
  Dadurch ist das System plattform-unabhängig und in der Lage mit jeder Java-Anwendung zu kommunizieren und zu interagieren.
- **Java Content Repository**
  Das System basiert auf der zukünftigen Standardschnittstelle für Content Management Systeme.
- **Schnelles Zwischenspeichern**
  Magnolia besitzt einen eigenen Zwischenspeicher, der in das Dateisystem geschrieben wird.
- **Internationalisierung**
  Durch die Unterstützung von Unicode ist es möglich jede Zeichensatzkodierung zu benutzen.
- **Rollenbasierte Benutzerverwaltung**
  Die Benutzerverwaltung speichert alle Information im *Java Content Repository*.
- **Meta-Daten Verwaltung**
  Magnolia speichert automatisch Meta-Daten, wie Autor, Erstellungsdatum und andere.
- **Synchronisation und Staging**
  In einer typischen Magnolia Installation werden die Inhalte vom Authoring-System zu einem öffentlichen System übertragen. Der eingebaute *Syndication*-Mechanismus erlaubt den Austausch von Inhalten mit anderen Anwendungen, die über HTTP kommunizieren können.

3.5 Bewertung

Die *Content Repository API for Java Technology* (JCR) ist dabei ein erster Schritt zur Standardisierung von Diensten der Repository-Technologie. Die Ausarbeitung einer Spezifikation schreitet jedoch nur langsam voran. Nach dem der angestrebte Termin im Juli 2003 für eine endgültige Version der Spezifikation nicht gehalten werden konnte, wurde er um ein Jahr verschoben. Ob dieser Termin haltbar ist darf angezweifelt werden, da einige Meilensteine auf dem Weg dorthin noch nicht erreicht wurden. Über einige wichtige offene Punkte, wie Versionsverwaltung und Anfragesprachen, scheint noch Uneinigkeit zu herrschen und eine Lösung dafür ist noch nicht in Sicht. Nichts desto trotz enthält JCR aber einige

interessante Gesichtspunkte. Der Entwurf der Schnittstelle zum Zugriff auf Repository-Objekte sowie die Semantik zur Manipulation dieser Objekte erscheinen gut gelungen.

Es bleibt abzuwarten, in wieweit Dinge, die im Moment noch unausgereift scheinen, wie die Datendefinitionssemantik und -schnittstellen, in die endgültige Version der Spezifikation einfließen. Hier stellt sich die Frage, warum nicht auf XML Schema als Sprache zu Datendefinition zurückgegriffen wurde. Trotzdem sollte die Datendefinitionssemantik mächtig genug sein, um viele vorhandene Content Management Repositories durch die JCR Schnittstelle abbilden zu können.

Da die JCR Spezifikation alle Anforderungen, die an eine Repository-Schnittstelle gestellt werden erfüllt, wird nun im Folgenden versucht die Integration von heterogenen Informationssystemen in die CoreMedia Smart Content Technology mit Hilfe von JCR vorzunehmen.

4 CoreMedia Smart Content Technology

Die CoreMedia *Smart Content Technology*, kurz SCT genannt, ist ein Content-Management-System. Es unterstützt die Erfassung und Eingabe von Inhalten, wie Texten und Bildern, die Qualitätskontrolle in Form des Vier-Augen-Prinzips und die Auslieferung von Inhalten an eine Vielzahl von unterschiedlichen Endgeräten, wie Computern, Mobiltelefonen, PDAs und Settop-Boxen.

4.1 Architektur

Die Architektur der *Smart Content Technology* ist eine ereignisbasierte Komponentenarchitektur (siehe Abbildung 11), deren Elemente hier kurz vorgestellt werden. Die Komponenten werden dabei in ein Produktions- und ein Live-System unterteilt. Ersteres dient der Erfassung und Pflege der Inhalte. Diese werden nach abgeschlossener Qualitätssicherung an das Live-System übertragen und so veröffentlicht. Dabei wird die Struktur der zu publizierenden Dokumente überprüft, um zum Beispiel Verweise auf nicht existierende Ressourcen zu vermeiden. Die Server auf der Seite des Live-Systems sind nach dem Master/Slave-Prinzip aufgebaut. Es gibt einen Master Live Server, der die Daten vom Produktionssystem empfängt und eine große Anzahl von Slave Live Servern, die ihre Daten vom Master Live Server erhalten.

Die Kommunikation zwischen den Servern sowie zwischen dem Server und den Klienten wird über CORBA [COR02] abgewickelt. Dadurch ist es möglich jederzeit neue Komponenten über diese Software-Schnittstelle anzubinden. Die Inhalte werden in der Regel über das HTTP-Protokoll [FGM+99] an die Endgeräte ausgeliefert.

- **Content Server:** Dieses ist die zentrale Komponente zur Verwaltung der Inhalte auf der Produktionsseite, die durch die Editoren und Importer in das System eingearbeitet wurden. Es unterstützt eine versionierte Ablage von Dokumenten in einer Verzeichnisstruktur. Die Daten werden in einem relationalen Datenbanksystem persistent gehalten.

- **Workflow Server:** Der Workflow Server erlaubt die Abbildung von inhaltsbezogenen Geschäftsprozessen. Er unterstützt die Koordination von Aufgaben und Abläufen und kann so die Produktivität der Benutzer erhöhen, da sie sich nicht mit organisatorischen Aufgaben befassen müssen.

- **Importer:** Die Importer werden dazu genutzt, um Daten aus Fremdsystemen in ein einheitliches Format zu konvertieren und in das Produktionssystem zu integrieren. Dazu kann ein Importer für ein Dateiformat und eine Quelle konfiguriert werden, um diese Dateien in einem Eingangsverzeichnis im Content Server abzulegen.

- **Editor:** Die am Prozess der Content-Erstellung beteiligten Personen arbeiten in der Regel mit dem SCT Editor. Dieser bietet eine intuitive graphische Benutzteroberfläche zur Erzeugung und Pflege der im System vorhandenen Daten. Weiterhin ermöglicht der Editor die Administration des Systems über die integrierte Benutzer- und Rechteverwaltung.

- **Preview Generator:** Diese Komponente dient zur Vorschau der erstellten Inhalte auf dem Produktionssystem. Der Content kann so bereits in der Form betrachtet werden, in der er später auf dem Live-System veröffentlicht würde.
- **Master Live Server:** Nachdem die Qualitätskontrolle auf dem Produktionssystem abgeschlossen wurde, werden die neuen und veränderten Inhalte an das Live-System übertragen. Dieser Vorgang wird auch „publizieren“ oder *Staging* genannt. Der Master Live Server nimmt diese Inhalte entgegen und ermöglicht deren Repliktation auf die Slave Live Server.
- **Slave Live Server:** Diese Server halten exakte Kopien der Daten des Master Live Servers. Dieses dient der Skalierung bei der Auslieferung der Inhalte. Die Server gleichen sich hierzu automatisch mit dem Master Live Server ab.
- **Active Delivery Server:** Die *Active Delivery Server* liefern die veröffentlichten Inhalte an die unterschiedlichen Endgeräte aus. Pro Slave Live Server werden in der Regel mehrere *Active Delivery Server* verwendet, um die Leistungsfähigkeit des Systems zu steigern.
- **Proactive Delivery Server:** Dieser Server ist dafür zuständig die Inhalte in verschiedenen Ansichten darzustellen. Die Darstellung wird neu berechnet, sobald neue Inhalte publiziert werden.

4.2 SCT Server

Der SCT Server ist die zentrale Komponente in jedem SCT System. Es gibt mindestens zwei Instanzen des Servers, den *Content Server* auf der Produktionsseite und den *Master Live Server* auf der Auslieferungsseite. Hinzukommen je nach Konfiguration noch mehrere *Slave Live Server*. Ein SCT Server ist über JDBC [EH01] mit einem relationalen Datenbank-Managementsystem verbunden. Er speichert und verwaltet die Inhalte.

Der Server nutzt zur Kommunikation mit den Klienten CORBA. Der *Master Live Server* ist ein Klient des *Content Servers* und die *Slave Live Server* sind Klienten des *Master Live Servers*. Für die Kommunikation zwischen den Servern wird also keine separate Schnittstelle benötigt.

Der SCT Server beinhaltet und verwaltet das SCT Repository, in dem die Inhalte gespeichert werden.

4.2.1 Das Datenmodell

Inhalte werden im SCT Server als Ressourcen modelliert. Wie Abbildung 12 zeigt, ist eine Ressource entweder ein Verzeichnis, welches weitere Ressourcen enthalten kann oder ein versioniertes Dokument, dessen Eigenschaften die eigentlichen Inhalte enthalten. Ein ausgezeichnetes Verzeichnis, das Wurzelverzeichnis, bildet den Anfang der Verzeichnishierarchie. Die Versionen eines Dokuments bilden eine lineare Liste, das heißt jede Version, außer der ersten und der letzten, hat genau eine Vorgänger- und genau eine Nachfolgerversion.

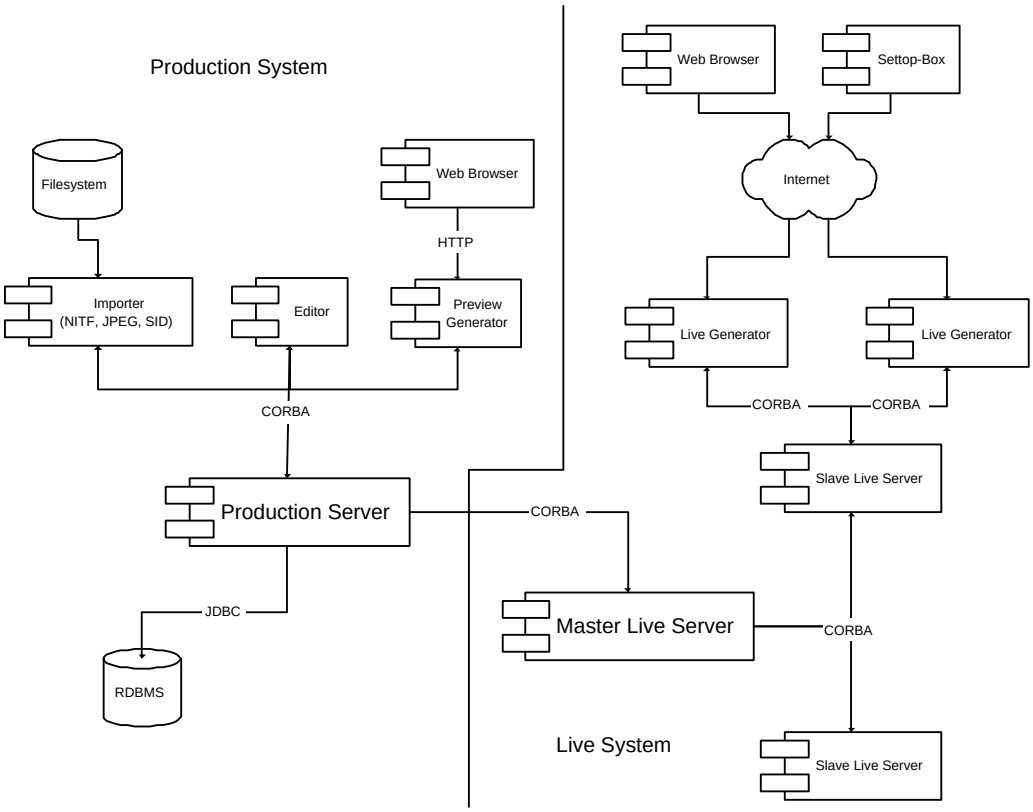

Abbildung 11: Die Architektur der Smart Content Technology

Ein Dokument hat genau einen Typ, welcher über die Dokumenttypdefinition (DTD) festgelegt wird. Dokumenttypen sind Mitglieder in einer Vererbungshierarchie. Ein Typ besitzt maximal einen Supertyp, wodurch ein Wald von Dokumenttypen beschrieben wird. Ein Dokumenttyp definiert eine Menge von Eigenschaftstypen. Ein Dokumenttyp vererbt die Eigenschaftstypen an seine Subtypen. Dokumenttypen können als „abstrakt“ definiert werden, was bedeutet, dass es kein Dokument geben kann, welches primär diesen Typ besitzt. Es ist aber möglich Dokumente eines Subtyps eines abstrakten Typen zu erzeugen. Der Typ eines Dokuments wird zum Erzeugungszeitpunkt festgelegt und lässt sich später nicht mehr ändern.

Auch die Eigenschaften besitzen Typen. Es gibt die sechs fest vorgegeben Eigenschaftstypen *String*, *Integer*, *XML*, *Date*, *BLOB* und *LinkList*. Für den Typ *XML* muss pro Eigenschaft eine Grammatik angegeben werden. Eine Grammatik ist in diesem Fall eine *XML Document Type Definition*. Die in der Eigenschaft gespeicherten Werte müssen dann gegen diese Grammatik validierbar sein.

Eine Version eines Dokuments befindet sich, vereinfacht dargestellt, in genau einem von fünf Zuständen. Diese werden in Abbildung 14 gezeigt. Direkt nach dem Erzeugen befindet sich eine Version im Zustand „ausgeliehen“, in dem sie vom Erzeuger bearbeitet werden kann. Danach durchläuft sie, wenn die zugehörigen Aktionen ausgeführt werden, die

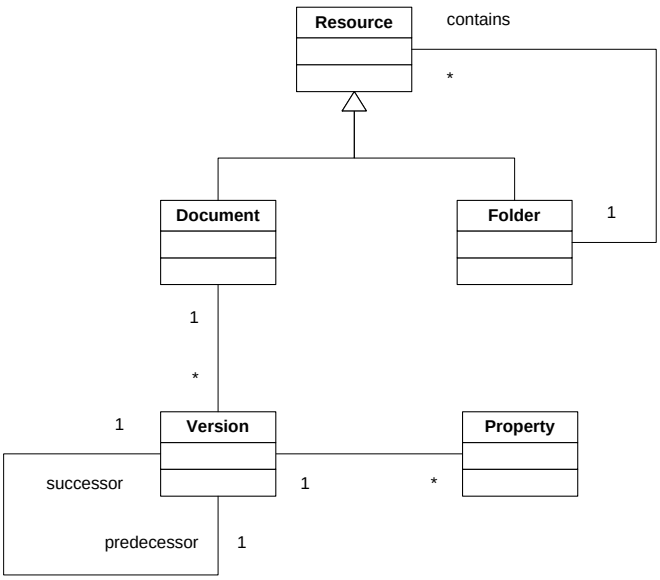

Abbildung 12: Modellierung der Inhalte in der Smart Content Technology

Zustände „zurückgegeben“, „freigegeben“ und „publiziert“. In jedem dieser drei Zustände kann die Version zerstört werden, wodurch sie in den Zustand „zerstört“ gelangt, welcher ein Endzustand ist. Eine Version kann nicht mehr bearbeitet werden, das heißt ihre Eigenschaften können nicht mehr verändert werden, wenn sie den Zustand „ausgeliehen“ verlassen hat. Um das Dokument trotzdem weiter bearbeiten zu können, muss eine neue Version angelegt werden.

4.3 SCT Active Delivery Server

Die Hauptaufgabe des *Active Delivery Server* ist es, Seiten zu generieren, welche über das HTTP-Protokoll ausgeliefert werden und deren Inhalte aus einem SCT Server stammen. Das Format der Seiten ist beliebig. Es kann sich dabei zum Beispiel um HTML-Seiten, PDF-Dokumente oder JPEG-Bilder handeln.

Der *Active Delivery Server* nutzt dazu Technologien wie die Programmiersprache Java [GJSB00], *Java Servlets* [Cow01], *Java Server Pages* (JSP) [RPL03], HTTP [FGM+99] und XML [BPSMM00].

4.3.1 Identifizierung von Ressourcen

Dokumente oder Seiten werden im *Active Delivery Server* eindeutig über *Uniform Resource Locators (URLs)* [BLMM94] identifiziert. Dazu werden alle URLs als `ResourceUri`-Objekte repräsentiert, welche alle Informationen enthalten, die notwendig sind um Seiten zu generieren. Ein `ResourceUri`-Objekt besteht aus genau fünf Attributen:

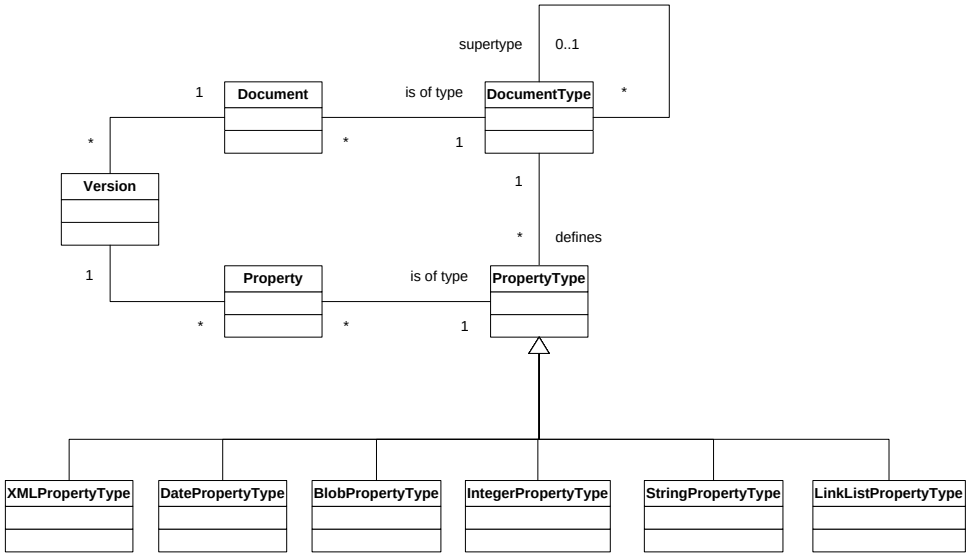

Abbildung 13: Das Metadatenmodell der Smart Content Technology

- **Ressource-ID**. Da eine Seite in der Regel primär eine bestimmte SCT Ressource anzeigt, identifiziert die Ressource-ID zu einem Teil die Seite. Natürlich kann die Seite auch weitere Ressourcen anzeigen, aber die primäre Ressource wird für eine einfachere Handhabung benutzt.
- **Name der Vorlage**. Um eine Seite darzustellen, wird eine Vorlage benutzt, in welche die Inhalte aus einer SCT Ressource eingefügt werden.
- **Versionsnummer**. Da Dokumente im SCT Repository versioniert sind, müssen alle Versionen eines Dokuments adressierbar sein.
- **Name einer Eigenschaft**. Um eine Eigenschaft einer Ressource direkt anzusprechen, kann der Name der Eigenschaft in der URL enthalten sein. Ein Anwendungsfall hierfür ist die Auslieferung eines Bildes, welches in einer BLOB-Eigenschaft eines Dokuments gespeichert ist.
- **Name-Wert-Paare**. Um eine Seite zu parametrisieren, kann die URL beliebige Name-Wert-Paare enthalten.

Da die Zeichenketten-Darstellung eines `ResourceUri`-Objektes anwendungsspezifisch ist, kann die Umwandlung von `ResourceUri`-Objekten in Zeichenketten und zurück frei konfiguriert werden.

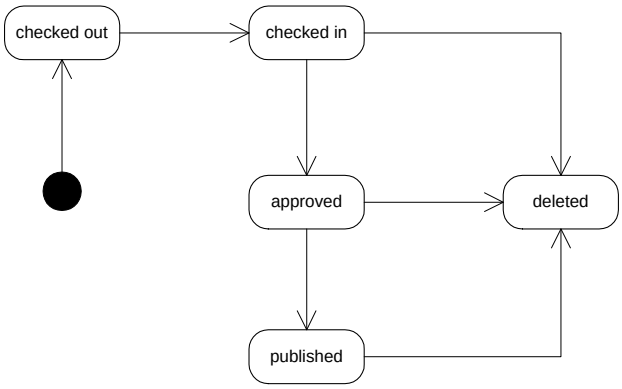

Abbildung 14: Vereinfachtes Zustandsmodell einer SCT Dokumentversion

4.3.2 Zuweisung einer Vorlage

Um eine Seite zu generieren, muss zunächst die Vorlage bestimmt werden, die dazu benutzt werden soll. Dafür ist eine Implementierung der Schnittstelle `TemplateFinder` zuständig, die aus einer gegebenen `ResourceUri` den Pfad einer JSP-Datei berechnet.

Ein bereits von CoreMedia mitgelieferter `TemplateFinder` ist der `ViewDispatcher`, welcher die Abbildung von Vorlagenname zu JSP-Datei anhand der SCT Dokumenttypen vornimmt. Dabei wird im Besonderen die Vererbungshierarchie der Dokumenttypen berücksichtigt.

Der `ViewDispatcher` basiert auf dem Konzept, dass Vorlagen als Ansichten von Dokumenten verstanden werden. Eine Vorlage dient dazu eine oder mehrere SCT Ressourcen als HTML-Seite, PDF-Dokument oder ähnlichem anzuzeigen. Dieses wird als Ansicht einer Ressource bezeichnet. Es kann auch Varianten einer Ansicht geben. Zum Beispiel kann ein Artikel in einer Kurzform auf einer Übersichtsseite angezeigt werden, als Suchergebnis oder auf einer ganzen Seite.

Die Eigenschaften eines Dokuments werden in der Dokumenttypdefinition festgelegt. Daher kann eine Vorlage einheitlich alle Dokumente eines bestimmten Typs anzeigen. Da Dokumenttypen eine Subtyp-Beziehung eingehen können, in der ihre Eigenschaften an den Subtyp vererbt werden, kann eine Vorlage auch Dokumente eines spezielleren Typs anzeigen. Dabei werden speziellere Eigenschaften des Subtyps nicht in Betracht gezogen.

Der `ViewDispatcher` interpretiert den Namen der Vorlage aus der `ResourceUri` als Ansicht in der die adressierte Seite angezeigt werden soll. Die Vorlage, die letztendlich zur Berechnung der Seite benutzt wird, ergibt sich aus dem Namen der Vorlage und dem Typ der anzuzeigenden Ressource. Findet der `ViewDispatcher` keine Vorlage für den Typ der anzuzeigenden Ressource, sucht er eine Vorlage für den Supertypen des Typen der Ressource. Das führt zu einer Vererbung von Ansichten, parallel zu der Vererbung von Eigenschaften.

4.3.3 Zwischenspeicherung von Seiten

Der Ausdruck Zwischenspeicher oder auch Cache ist aus dem Gebiet der Hardware bekannt [Smi82]. Ein Prozessor benutzt einen Zwischenspeicher um den Zugriff auf den Hauptspeicher zu beschleunigen. Im Allgemeinen wird ein Cache durch folgende Punkte charakterisiert:

- Die **Objekte**, die in dem Cache gespeichert werden. Im Fall des Prozessors sind dieses Speicherworte.
- Die **Schlüssel**, welche die Objekte im Zwischenspeicher identifizieren. In dem Beispiel vom Prozessor sind dieses die Speicheradressen.
- Der **externe Zustand** (der Inhalt der Zellen des Hauptspeichers des Prozessors), von dem die Gültigkeit eines Cache-Eintrags abhängt.
- Eine deterministische **Funktion**, die immer das gleiche Objekt für den gleichen Schlüssel liefert, vorausgesetzt, dass sich der externe Zustand nicht geändert hat.
- Ein **Benachrichtigung**s-Mechanismus, welcher den Zwischenspeicher informiert, wenn sich der externe Zustand ändert. Bei dem Fall mit dem Prozessor wäre dieses zum Beispiel ein *Bus Snooping* Protokoll.
- Die **Größe des Zwischenspeichers** und eine **Ersetzungsstrategie**, die bestimmen, welche Objekte im Cache gehalten werden.

Der Zwischenspeicher des *Active Delivery Servers* kann an Hand dieser Kriterien beschrieben werden:

- Die **Objekte** im Zwischenspeicher sind die generierten Seiten.
- Die **Schlüssel**, welche die Seiten im Zwischenspeicher identifizieren, sind `ResourceUri`-Objekte.
- Der **externe Zustand** ist in diesem Fall der Zustand des SCT Repositorys, von dem Inhalt der generierten Seiten abhängt. Ein Mechanismus um zu verfolgen, auf welche Ressourcen im Repository zugegriffen wurde, ist ebenfalls vorhanden und wird weiter unten in Abschnitt 4.3.4 beschrieben.
- Die Vorlagen sind die **Funktionen**, welche die Schlüssel in die Cache-Objekte überführen. Sie müssen für eine `ResourceUri` immer dieselbe Seite erzeugen.
- Da der *Content Server* die angebundenen Klienten über Änderungen am Repository **benachrichtigt**, kann der *Active Delivery Server* zu jedem Zeitpunkt feststellen, ob ein Cache-Eintrag gültig ist.
- Die **Größe des Zwischenspeichers** kann in der Konfigurationsdatei des *Active Delivery Servers* eingestellt werden. Als **Ersetzungsstrategie** wird immer ein LRU-Algorithmus verwendet, der immer die Seite aus dem Zwischenspeicher entfernt, die am längsten nicht ausgeliefert wurde.

4.3.4 Verfolgung von Abhängigkeiten

Um die Objekte im Zwischenspeicher effizient und korrekt zu invalidieren, wenn sich der Zustand des SCT Repositorys ändert, muss der *Active Delivery Server* verfolgen, auf welche Ressourcen zugegriffen wurde, während eine Seite erzeugt wurde. Es werden zwei Typen von Abhängigkeiten unterschieden, direkte und abgeleitete Abhängigkeiten.

Direkte Abhängigkeiten

Direkte Abhängigkeiten werden erzeugt, wenn Objekte außerhalb des *Active Delivery Servers* benutzt werden. Um die Gültigkeit dieser Abhängigkeiten sicher stellen zu können, muss der *Active Delivery Servers* über Änderungen an diesen externen Objekten benachrichtigt werden.

Abgeleitete Abhängigkeiten

Abgeleitete Abhängigkeiten werden erzeugt, wenn ein Objekt benutzt wird, welches der *Active Delivery Servers* selbst erzeugt hat. Im Prinzip ist eine abgeleitete Abhängigkeit die Summe der Abhängigkeiten des benutzten Objekts, welche transitiv zu den direkten Abhängigkeiten zurückverfolgt werden können.

4.4 SCT Proactive Delivery Server

Der *Proactive Delivery Server* generiert Seiten, sobald die Inhalte dafür verfügbar sind, wie zum Beispiel nach einem Publikationsvorgang. Dies ist ein grundlegender Unterschied zum *Active Delivery Server*, welcher Seiten erst generiert, wenn eine Anfrage dafür gestellt wurde.

Der *Proactive Delivery Server* ist nicht auf den Seitenbau durch Vorlagen beschränkt. Es können beliebige Ausgaben, die auf SCT Ressourcen basieren, erzeugt werden und dabei ein effizienter Zwischenspeicher und die Verfolgung von Abhängigkeiten benutzt werden, wie sie von *Active Delivery Server* bekannt sind.

Der *Proactive Delivery Server* benutzt ein relationales Datenbanksystem, um den Cache persistent zu halten. Das bietet den Vorteil, dass der Inhalt des Zwischenspeichers nicht neu berechnet werden muss, wenn das System neu gestartet wird.

4.5 SCT Web Application Generator Extensions

Die SCT *Web Application Generator Extensions*, kurz WAGE, sind ein Baukasten für Web Anwendungen, welche die Bearbeitung von SCT Ressourcen unterstützen und Workflow-Funktionalität bieten sollen.

WAGE erweitert die Programmierschnittstelle des *Active Delivery Servers*, welches nur lesenden Zugriff auf das SCT Repository erlaubt, um einen Schreib-Zugriff sowie um den Zugriff auf den Workflow-Server.

Das WAGE Rahmenwerk besteht aus server- und klientenseitigen Komponenten. Die Komponenten im Server basieren auf der Programmierschnittstelle des *Active Delivery Servers*

und des Jakarta Struts Projektes, während die klientenseitigen Teile mit DHTML und JavaScript realisiert sind.

Die *Web Application Generator Extensions* laufen in dem SCT *Authoring Server*, welcher ein spezieller *Active Delivery Server* ist.

Ein großer Vorteil des *Authoring Servers* ist die Unterstützung für eine sehr große Anzahl von Benutzern, die über ein Web-Frontend SCT Ressourcen bearbeiten und an Workflows teilnehmen können. Der SCT Editor öffnet für jeden Benutzer jeweils eine Verbindung zum Content Server sowie eine zum Workflow Server. Änderungen, die der Benutzer an Ressourcen durchführt, wie das Ändern einer Eigenschaft oder das Erzeugen eines neuen Ordners, sind sofort für alle anderen Benutzer sichtbar. Bei einer Web Anwendung, die auf WAGE basiert, wie der SCT WebEditor, benutzen die Anwender einen Browser, der mit dem Authoring Server über HTTP kommuniziert. Alle Änderungen, die der Benutzer durchführt, werden auf dem Authoring Server gesammelt und als Menge zum Content Server geschickt. Trotzdem ist der Benutzer in der Lage, eine Voransicht über die Ergebnisse seiner Änderungen zu bekommen. Der Authoring Server benötigt nur eine Verbindung zum Content Server und zum Workflow Server. Für den eigentlichen Benutzer werden leichtgewichtige Verbindungen benutzt, die auf dem Content Server weniger Ressourcen erfordern als normale Verbindungen. So wird die Last auf dem Content Server reduziert und die Skalierbarkeit erhöht.

4.5.1 Das Struts Rahmenwerk

Das Struts Rahmenwerk aus dem Jakarta Projekt der Apache Software Foundation unterstützt eine 3-Schichten Architektur für Web Anwendungen. Es basiert auf dem *Model 2* Ansatz [Ses99], welcher eine Variation des klassischen *Model-View-Controller (MVC)* Entwurfparadigmas ist.

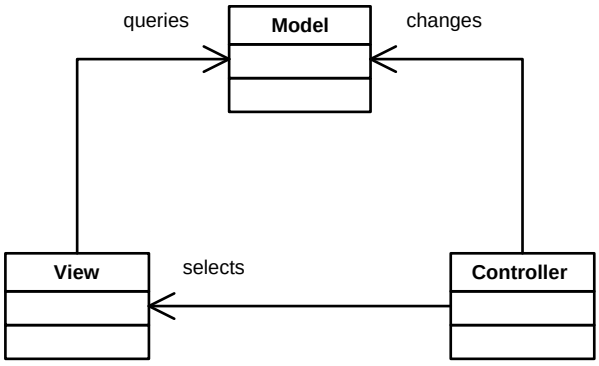

Abbildung 15: Das Model-View-Controller Entwurfsmuster

Der Controller des MVC Entwurfmusters ist für den Kontrollfluss innerhalb der Anwendung zuständig. Er leitet Anfragen (z.B. HTTP-Anfragen) an Steuerungsprogramme weiter. Diese

Steuerungsprogramme sind an ein Modell gebunden und agieren als Adapter zwischen den Anfragen und dem Modell. Ein Modell kapselt die Geschäftslogik oder den Zustand einer Anwendung. Nach der Behandlung einer Anfrage wird der Kontrollfluss in der Regel durch den Controller an eine Ansicht (*View*) weitergeleitet.

Bei dem *Model 2* Ansatz sind *Java Servlets* im Allgemeinen für den Kontrollfluss zuständig, wohingegen *Java Server Pages (JSPs)* für die Generierung von HTML-Code genutzt werden.

Das Modell kann in zwei Subsysteme unterteilt werden. Ein Teil hält den inneren Zustand der Anwendung und der andere Teil ist für die Aktionen verantwortlich, die nötig sind, um den Zustand zu ändern. Der innere Zustand wird häufig durch eine Menge von *Java Bean* Objekten [Ham96], auch *Business Objects* genannt, repräsentiert, wobei die Eigenschaften der *Beans* die Details des Anwendungszustands enthalten. Die Aktionen oder auch *Business Operations* werden durch Methoden implementiert, die auf den *Beans*, welche den Zustand verwalten, aufgerufen werden können.

Die JSP-Vorlagen (*Templates*) werden als Ansicht benutzt und enthalten keine Geschäftslogik und haben auch kein Wissen über den Kontrollfluss.

Um die Entwicklung von SCT Web Anwendungen zu vereinfachen, erweitert WAGE das Struts Rahmenwerk in dem es SCT-spezifische Implementierungen für diverse Struts Schnittstellen und Erweiterungen für Struts Klassen mit SCT-Unterstützung bietet. Darauf aufbauend können dann Web Anwendungen für den SCT *Authoring Server* entwickelt werden.

4.5.2 Der ChangeCollector-Mechanismus

Um in der Lage zu sein, Ressourcen zu ändern ohne die Änderungen sofort an den Server zu schicken, führt WAGE das Konzept der `ClientResource` ein. Eine `ClientResource` ist eine Ressource, die lokal auf dem Klienten modifiziert werden kann. Alle Änderungen sind für den Klienten sofort sichtbar, aber für alle weiteren Klienten erst nachdem die Menge der Änderungen zum Server geschickt wurde. Außerdem können `ClientResources` neue Ressourcen darstellen, welche noch nicht auf dem Server angelegt wurden. Die Ressource werden auf dem Server erst angelegt, wenn die aktuelle Menge von Änderungen abgeschickt wurde.

Das Objekt, welche die lokalen Änderungen verwaltet, wird `ChangeCollector` genannt. Jedem Klienten ist mindestens ein `ChangeCollector` zugeordnet. Klienten teilen sich aber niemals einen `ChangeCollector`. Da die `ChangeCollectors` untereinander nicht in Beziehung stehen, ist es möglich, dass unterschiedliche Klienten zueinander inkompatible Änderungen vornehmen. Zum Beispiel verlinkt Klient A ein Dokument, während Klient B dieses löscht. Diese Inkompatibilitäten werden erst nach dem Abschicken auf dem Server festgestellt und dem Klienten gemeldet. Der Vorgang der Übermittlung der Änderungen läuft nicht transaktional ab. Das bedeutet, dass alle Änderungen, die ausgeführt werden können, auch ausgeführt werden. Änderungen, die Fehler hervorrufen, werden gesammelt und an den Klienten zurückgeschickt.

Das WAGE Rahmenwerk verbirgt diese Funktionalität zum größten Teil, sodass es für den Entwickler transparent ist.

4.6 SCT Workflow Server

An großen Mehrbenutzersystemen, wie der CoreMedia Smart Technology, arbeiten viele Benutzer in unterschiedlichen Rollen. Je mehr Nutzer Inhalte bearbeiten, freigeben und publizieren, desto größer ist der Koordinationsaufwand. Eine Rechnerunterstützung kann die Produktivität in diesem Bereich erhöhen. Dies Ziel kann durch die Einführung eines Workflow-Systems erreicht werden.

Der SCT Workflow Server bietet die Möglichkeit inhaltsbezogene Geschäftsprozesse mit mehreren Benutzern und Ressourcen zu modellieren. Diese Prozesse werden durch Workflow Definitionen beschrieben. Eine Workflow Definition besteht aus einer Reihe von Aufgabendefinitionen und der Spezifizierung des Kontrollflusses zwischen den Aufgaben. Der SCT Workflow Server erlaubt sequenzielles, nebenläufiges, synchronisierbares, iteratives und bedingtes Ausführen von Aufgaben. Aufgaben können mit Eingangs- und Ausgangsaktionen versehen werden. So können Berechnungen, Benachrichtigungen und Eskalationsschritte implementiert werden.

Der SCT Workflow Server kann durch eine ereignisbasierte Programmierschnittstelle von externen Systemen gesteuert werden. Ebenso können Dritt-Systeme in den Workflow Server integriert werden. Eine Anbindung an den SCT Content Server ist bereits vorhanden.

Eine brauchbare Notation für die Definition von Workflows sind Ablaufdiagramme, wie sie in der *Unified Modeling Language* [UML03] spezifiziert sind. Die SCT Workflow Definitionen basieren auf den Ablaufdiagrammen, welche in ein spezielles XML-Format konvertiert werden müssen, damit sie vom Workflow Server verstanden werden.

5 Integration von CoreMedia SCT und Java Content Repository

In diesem Kapitel soll nun untersucht werden, wie heterogene Informationssysteme in ein bestehendes Content Management System integriert werden können. Als Schnittstelle zu den Informationssystemen soll, aus den bereits erläuterten Gründen, das *Java Content Repository* API verwendet werden.

Diese Programmierschnittstelle sieht aber keine Konzepte für die Trennung von Produktionssystem und Live-System vor. Deshalb müssen solche noch entwickelt werden. Dieses betrifft besonders das so genannte *Staging*, also den Vorgang der Übertragung der Inhalte vom Produktionssystem auf das Live-System. Dieser Abschnitt beim Publikationsprozess ist nicht trivial, wenn auch Daten aus externen Datenquellen in das *Staging* einbezogen werden sollen. Abschnitt 5.1 behandelt diese Problematik.

Weiterhin muss ein effizientes Zwischenspeichern der Daten aus den externen Quellen möglich sein. Dieses ist notwendig, da es nicht vertretbar ist, dass die zusätzliche Last, die auf den integrierten Systemen bei einer direkten Anbindung anfallen würde, diese Systeme verlangsamt. Für das Zwischenspeichern soll nach Möglichkeit, die vorhandene Technologie aus dem SCT *Active Delivery Server* wieder verwendet werden.

Die Abbildung der SCT Schnittstelle auf das JCR API wurde nicht entworfen und implementiert, da die *Java Content Repository* Spezifikation noch nicht in einer endgültigen Fassung vorliegt. Die vorläufigen Versionen der Spezifikation enthielten viele, teils schwerwiegende Änderungen an den Konzepten und der Programmierschnittstelle. Eine Implementierung dieses nicht stabilen APIs erschien nicht sinnvoll, da bei jeder Änderung einige Teile neu entwickelt werden müssten.

5.1 Konzepte zur Integration von Informationssystemen

Es bestehen mehrere Konzepte oder Architekturen zur Integration von Informationssystem in Content Management Systeme. Diese werden mit ihren wichtigsten Eigenschaften sowie ihren Vor- und Nachteilen im Folgenden vorgestellt.

Die Konzepte unterscheiden im Wesentlichen in dem Punkt, wie das zu integrierende System in den Staging-Prozess eingebunden wird. Da das Staging nicht direkt durch die JCR Spezifikation unterstützt wird, müssen hier eigene Betrachtungen durchgeführt werden.

5.1.1 Späte Integration

Werden die Daten aus einem Informationssystem durch die Inhalte im Content Management System referenziert, wird dieses eine virtuelle oder späte Integration genannt. Die eigentliche Integration geschieht bei der Präsentation der Daten.

Bei diesem Ansatz werden keine Daten kopiert. Um auf externe Daten zu verweisen, werden eindeutige Bezeichner, wie Primärschlüssel verwendet.

Zu den Vorteilen der späten Integration zählen:

- Im Gegensatz zur frühen Integration (siehe unten) ist weniger Speicherplatz notwendig, da keine Daten, bis auf Primärschlüssel, in das Content Management System kopiert werden.
- Es ist kein Abgleich zwischen *Content Server* und externem Informationssystem notwendig.
- Bei strukturellen Änderungen an dem externen Informationssystem muss das *Content*-Modell im Content Management System nicht geändert werden.
- Es ist nicht notwendig eine Komponente zum Abgleich zwischen CMS und externer Datenquelle zu entwickeln. Die Anbindung geschieht direkt in der Präsentationsschicht oder in einer Abstraktionsebene darunter.

Es treten aber auch einige Nachteile auf:

- Durch den direkten Zugriff ist ein Online-Zugriff auf die externe Datenquelle notwendig. Ist das Informationssystem durch eine Firewall geschützt, muss entweder ein Loch in die Firewall eingebracht werden oder das System muss auf der öffentlichen Seite der Firewall platziert werden. Beide Optionen stellen ein potentielles Sicherheitsrisiko dar.
- Es entsteht ein weiterer Punkt, an dem Performance-Engpässe auftreten können.
- Der Durchsatz des externen Informationssystems sinkt durch die zusätzliche Last, die durch die Integration auftritt.
- Durch die Integration eines weiteren Systems steigt die Wahrscheinlichkeit eines Ausfalls im Fehlerfall.
- Es besteht nur eine lose Kopplung der Systeme durch Verknüpfung über Primärschlüssel. Eine Sicherstellung der Integrität dieser Schlüssel über Systemgrenzen hinweg ist nur schwer möglich.

5.1.2 Frühe Integration

Die frühe Integration erlaubt eine Einbettung der Daten aus externen Informationssystemen in das CMS-Repository. Dabei werden die Daten direkt in das Content Management System kopiert. Dieser Ansatz erlaubt dem CMS eine volle Kontrolle über diese Daten.

Es ist möglich die Änderungen an den Daten, welche ursprünglich aus einem externen Informationssystem stammen, entweder inkrementell oder stapelweise in das Informationssystem zu übertragen.

Für eine nahtlose Integration muss eine Komponente existieren, welche Daten aus dem Schema des externen Informationssystems in das Schema des Content Management Systems überführt. Dieses Teilsystem wird in Abbildung 17 „*Data Mirror*“ genannt.

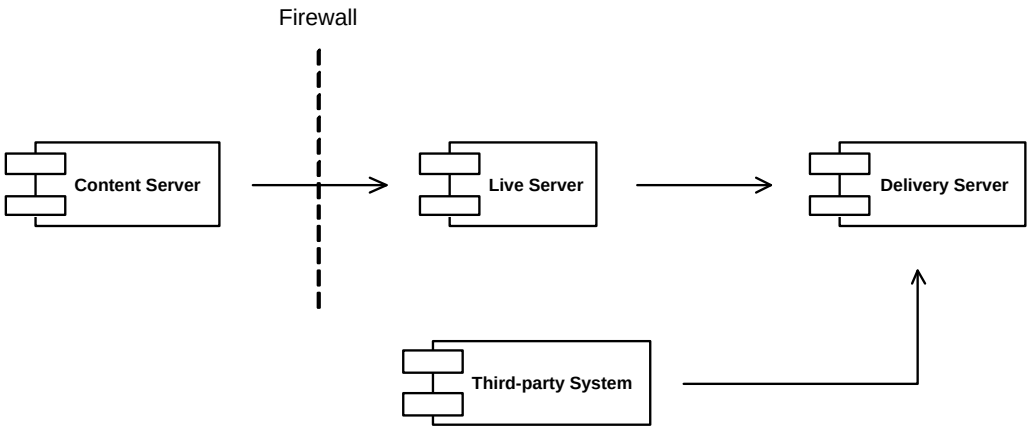

Abbildung 16: Architektur einer späten Integration

Die Vorteile einer frühen Integration sind:

- Es entsteht keine große Belastung des externen Informationssystems durch Seitenzugriffe.
- Es muss nur für ein System, nämlich dem Content Management System, eine 24×7 Laufzeit gewährleistet sein.
- Nur das Content Management muss mit der Anzahl der Zugriffe skalierbar sein.
- Die Verwaltung der Verknüpfungen wird durch das Content Management System übernommen. So besteht eine volle Kontrolle über die Verknüpfungen und es kann sichergestellt werden, dass keine Verknüpfungen existieren, deren Ziel nicht mehr gültig ist.

Auch bei der frühen Integration existieren einige Nachteile:

- Es wird mehr Speicherplatz benötigt, da alle Daten aus den externen Informationssystemen in das Content Management System kopiert werden.
- Strukturelle Änderungen an den Inhalten in den externen Informationssystemen müssen auch in das Content Management System übernommen werden.
- Die Komponente, welche die Verbindung zwischen Content Management System und externer Datenquelle herstellt, muss projektspezifisch und die jeweiligen Schemata des CMS bzw. des externen Informationssystems angepasst oder neu entwickelt werden.
- Es ist schwierig alle integrierten Systeme konsistent zu halten, wenn mindestens eines der beteiligten Systeme kein Transaktionskonzept unterstützt.

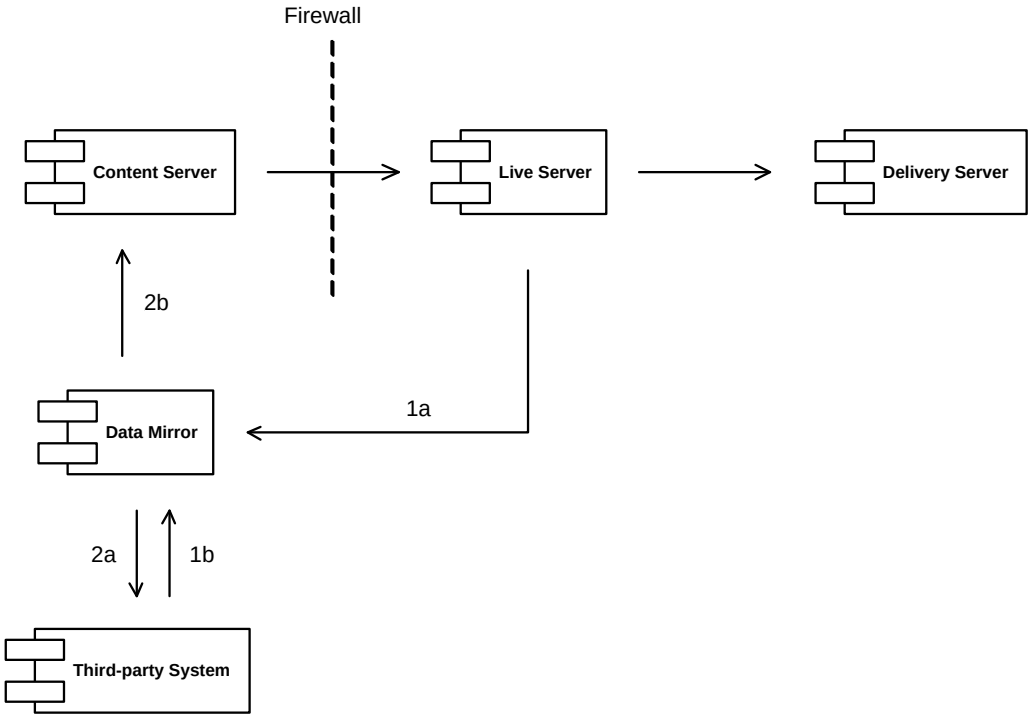

Abbildung 17: Architektur eine frühen Integration

5.1.3 Gemischte Integration

Bei der gemischten Integration werden Ansätze aus der frühen und der späten Integration genutzt, um die Vorteile und Nachteile aus beiden Konzepten zu kombinieren und so zu einer Lösung zu gelangen, die den Anforderungen am besten entspricht (siehe Abbildung 18.

Bei dieser Art der Integration ist es möglich, Inhalte, auf die nur selten zugegriffen wird oder die sich häufig ändern, in dem externen Informationssystem zu belassen. Daten, die stark abgefragt werden oder von einem Redakteur in Content Management System gepflegt werden sollen, können in das CMS kopiert werden.

5.2 Entwurf

5.2.1 Architektur zur Integration von Java Content Repositories

Die grobe Architektur für die Anbindung externer Informationssysteme besteht aus zwei Teilen, der Auslieferung und dem Authoring. Die Aufgabe der Auslieferungskomponente besteht in der effizienten Übermittlung der Inhalte an die Klienten des Content Management System, zum Beispiel die Besucher einer Web-Seite. Mit dem Begriff Authoring wird die Erstellung und Pflege von Inhalten in Content Management System bezeichnet.

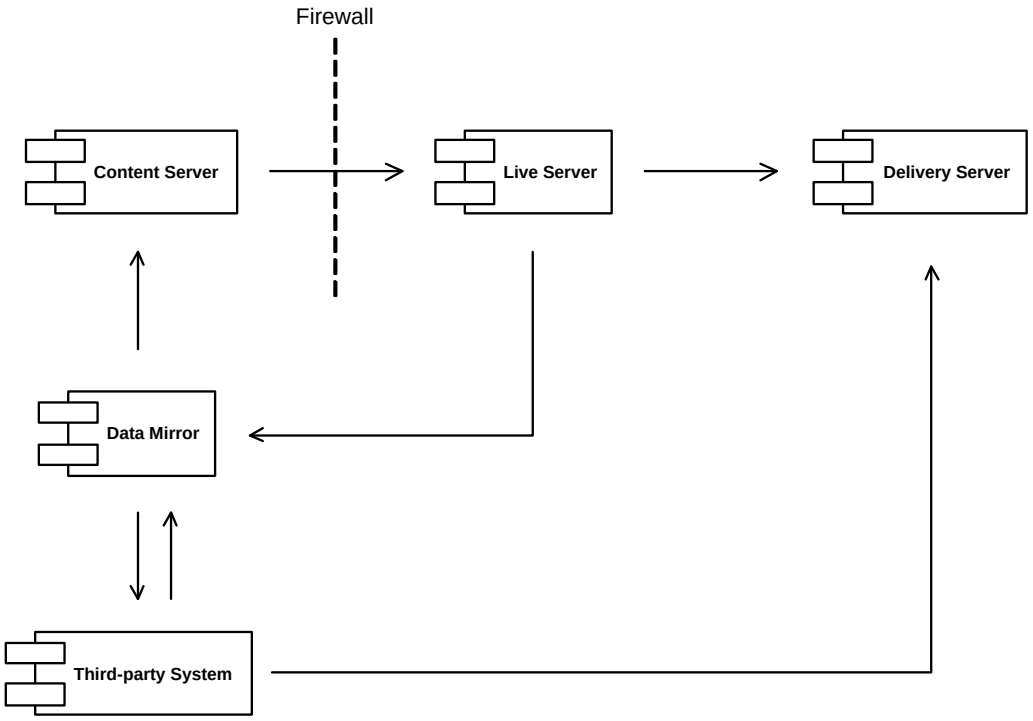

Abbildung 18: Architektur einer gemischten Integration

Die Trennung der Architektur wurde vorgenommen, um den unterschiedlichen Fähigkeiten der anzubindenden Repositories zu berücksichtigen. Außerdem sieht auch die Architektur der *CoreMedia Smart Content Technology* hier eine Teilung vor (siehe Abbildung 11).

Auslieferung

Dieser Teil der Implementierung befasst sich mit der Erzeugung von Seiten aus Inhalten und Vorlagen. Die Seiten werden für eine performante Auslieferung zwischengespeichert. Die bestehende Auslieferungskomponente soll um die Auslieferung von Inhalten aus externen Informationssystemen erweitert werden. Dadurch kann hier keine vollständig neue Architektur entwickelt werden. Die bestehenden Komponenten werden sinnvoll erweitert. Wie Abbildung 19 zeigt, ist JCR eine weitere Schnittstelle, neben der SCT CORBA API, über die auf Inhalte zugegriffen wird.

Für diesen Teil der Implementierung werden folgende Anforderungen an ein Repository gestellt:

- **Lesezugriff**
  Wird zum Auslesen und Durchsuchen der Elemente aus dem *Java Content Repository* benötigt.

- **Typisierung**
  Um eine objektorientierte Auffindung einer Vorlage zu ermöglichen wird die Unterstützung der Typisierung der Knoten gefordert.
- **Überwachung**
  Zur Invalidierung der Elemente im Zwischenspeicher wird Überwachung benötigt. Ist diese Funktionalität nicht durch das Repository implementiert, kann der Zwischenspeicher nicht genutzt werden.
- **Zugriffskontrolle**
  Optional. Kann zur Einschränkung des Zugriffs auf eine Web-Seite benutzt werden.
- **Suche**
  Optional. Kann zum effizienten Auffinden von Inhalten genutzt werden.

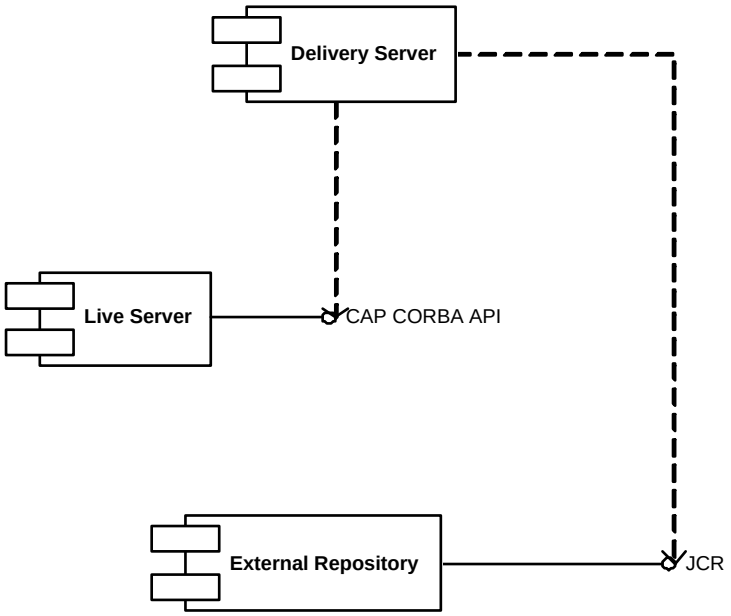

Abbildung 19: Architektur der Auslieferung von Inhalten aus externen Datenquellen

Authoring

Bei der Authoring-Komponente liegt der Fokus auf der Erstellung und Pflege von Inhalten. Zu diesem Zweck wurde das *Authoring Server Framework* (siehe Abschnitt 4.5) entwickelt, welches ein Rahmenwerk zur Erstellung von Formularen und Editoren bietet. Da dieses Rahmenwerk sehr gut erweiterbar ist, wird auch hier keine neue Komponente entwickelt, sondern die JCR Schnittstelle in das *Authoring Server Framework* integriert (siehe Abbildung 20).

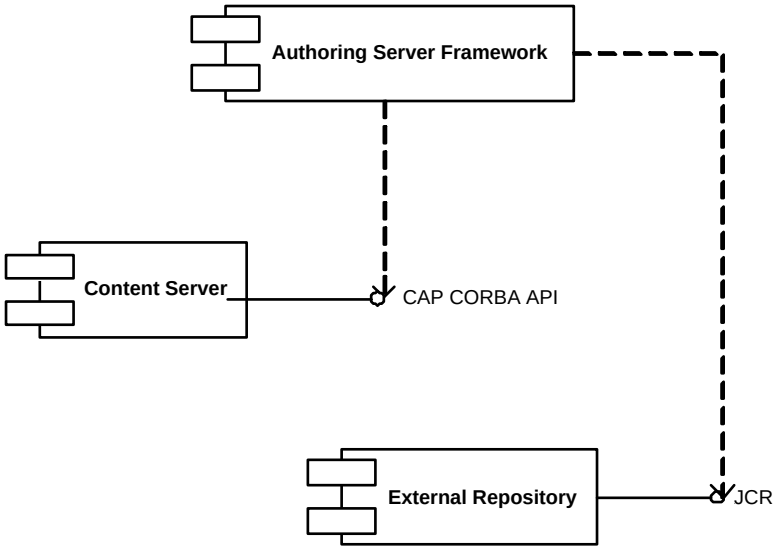

Abbildung 20: Architektur des Authoring von Inhalten aus externen Datenquellen

Die Anforderungen der Authoring-Komponenten an ein *Java Content Repository* ergänzen die Anforderungen der Auslieferung um folgende Punkte:

- **Schreibzugriff**
  Wird zum Speichern der erstellten Inhalte in das *Repository* benötigt.
- **Versionierung**
  Optional. Für ein reibungsloses Authoring in einer Mehrbenutzerumgebung notwendig.
- **Sperren**
  Optional. Ebenso wie Versionierung für ein reibungsloses Authoring in einer Mehrbenutzerumgebung notwendig.
- **Transaktionen**
  Optional. Notwendig um Inhalte in einer Mehrbenutzerumgebung konsistent zu halten.

5.3 Implementierung

In den folgenden Abschnitten wird auf die Details der Implementierung eingegangen. Dazu zählen die Adressierung von Ressourcen aus integrierten Informationssystemen, das Auffinden von Vorlagen zur Darstellung dieser Ressourcen, das Festlegen eines Formats für die Zeichenkettenrepräsentation einer URL, das Erzeugen von `Uri`-Objekten sowie die Verfolgung von Abhängigkeiten von und zwischen Ressourcen und das damit verbundene Zwischenspeichern.

5.3.1 Adressierung externer Ressourcen

Um die Seiten, die vom *Active Delivery Server* erzeugt wurden, über HTTP abzurufen, müssen diese eine eindeutige Adresse besitzen. Die Adresse wird auch als *Uniform Resource Identifier* oder kurz *URI* bezeichnet.

In der bisherigen Implementierung des *Active Delivery Servers* war es nur möglich Ressourcen aus dem SCT Repository zu adressieren. Als Basisschnittstelle für eine *URI* wird im *Active Delivery Server* die Schnittstelle `ResourceUri` verwendet (siehe Abbildung 21). Implementierungen dieser Schnittstelle enthalten Informationen über die adressierte SCT Ressource, die zu verwendende Vorlage, den Sicherheitskontext und die Zwischenspeicherung. Diese Implementierung ist nicht erweiterbar, da keine Schnittstelle vorhanden ist, die eine SCT-unabhängige Adressierung ermöglicht.

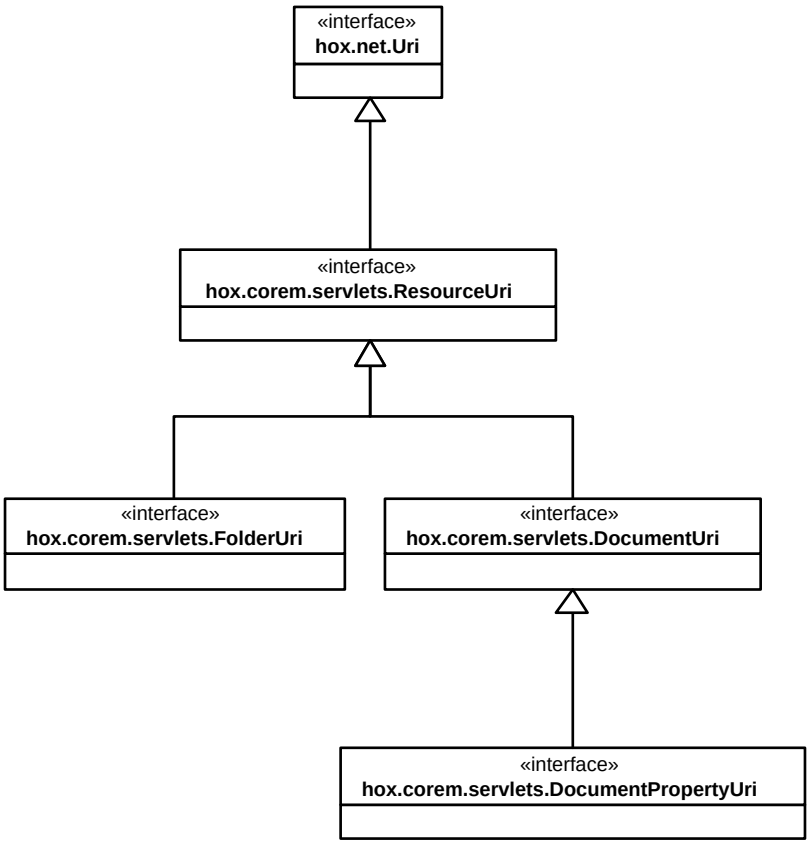

Abbildung 21: Klassendiagramm des Ist-Zustandes der Adressierung von Ressourcen

Der Adressierungsmechanismus im *Active Delivery Server* wurde dahingehend erweitert, dass dieser beliebige Ressourcen adressieren kann, also nicht nur Dokumente, Verzeichnisse und Eigenschaften des SCT Repositorys. Dazu wurde die Schnittstelle `ResourceUri` in zwei

Schnittstellen aufgeteilt. Dieses sind die in Abbildung 22 hellgrau dargestellten Schnittstellen. Es wurde eine neue Schnittstelle `Uri` eingeführt, welche genereller als `ResourceUri` ist und Informationen über die zu benutzende Vorlage, das Zwischenspeichern und den Sicherheitskontext liefern kann. Die Methoden dafür wurden aus `ResourceUri` exakt übernommen, sodass die neue `ResourceUri` Schnittstelle mit der alten kompatibel bleibt. Die neue Version der `ResourceUri` Schnittstelle erweitert die neue allgemeine `Uri` Schnittstelle um eine Referenz auf die adressierte SCT Ressource.

Eine weitere Verfeinerung der neu eingeführten, allgemeinen `Uri` Schnittstelle stellt die `JcrUri` Schnittstelle dar, welche `Uri` um einen Verweis auf das adressierte *Java Content Repository* Element erweitert. Um nicht nur ein *Java Content Repository* zu unterstützen, sondern beliebig viele, enthält die `JcrUri` auch eine Bezeichnung des anzusprechenden Repositorys.

5.3.2 Auffinden der Vorlagen

Um eine HTML-Seite oder ein PDF-Dokument aus einer oder mehreren SCT oder externen Ressourcen zu erzeugen, wird eine Vorlage benötigt, in die Inhalte eingefügt werden. Der Prozess des Auffindens von Vorlagen beschäftigt sich mit der Zuordnung einer Vorlage zu einer adressierten Ressource. Wie im vorhergehenden Abschnitt erklärt, wird durch eine *URI* genau eine Ressource adressiert. Diese Ressource ist dann die Hauptressource um eine Seite aus einer Vorlage zu generieren. Es ist natürlich möglich in der Vorlage weitere Ressourcen zu benutzen, um die Seite durch deren Inhalte zu ergänzen.

Um nun eine Vorlage für eine gegebene *URI* zu finden, wird die Schnittstelle `TemplateFinder` benutzt. Diese liefert für ein `ResourceUri`-Objekt den Namen einer Vorlage. Um einen konkreten Algorithmus zu implementieren muss eine Klasse mit der Schnittstelle `TemplateFinder` erstellt werden. Dieses entspricht dem *Strategy* Entwurfsmuster aus [GHJV95].

Wie Abbildung 23 zeigt, sind im *Active Delivery Server* bereits zwei Implementierungen der Schnittstelle `TemplateFinder` enthalten.

Der `StandardTemplateFinder` interpretiert die `templateId`-Eigenschaft des `ResourceUri`-Objektes als absoluten oder relativen Pfad zu der gewünschten Vorlage. Im Falle eines relativen Pfades wird eine Suche in den Verzeichnissen unterhalb des Vorlagenwurzelverzeichnisses durchgeführt. Der `StandardTemplateFinder` enthält dazu eine Menge von Regeln, die SCT Dokumenttypnamen und Eigenschaftsnamen auf Vorlagennamen abbilden. Die Suche läuft dann von dem Unterverzeichnis, welches dem SCT Verzeichnis der anzuzeigenden Ressource entspricht, bis zum Vorlagenwurzelverzeichnis mit dem Ziel eine Vorlage mit dem passenden Namen im lokalen Dateisystem zu finden.

Eine komplexere Auflösung der Vorlage erlaubt der `ViewDispatcher`. Der `ViewDispatcher` erweitert die `TemplateFinder`-Schnittstelle um das Konzept der Ansichten auf Dokumenttypen. Diese Ansichten sind die „Methoden“ eines Dokumenttyps, die als Vorlagen (in der Regel als *Java Server Pages*) implementiert sind. So ist eine

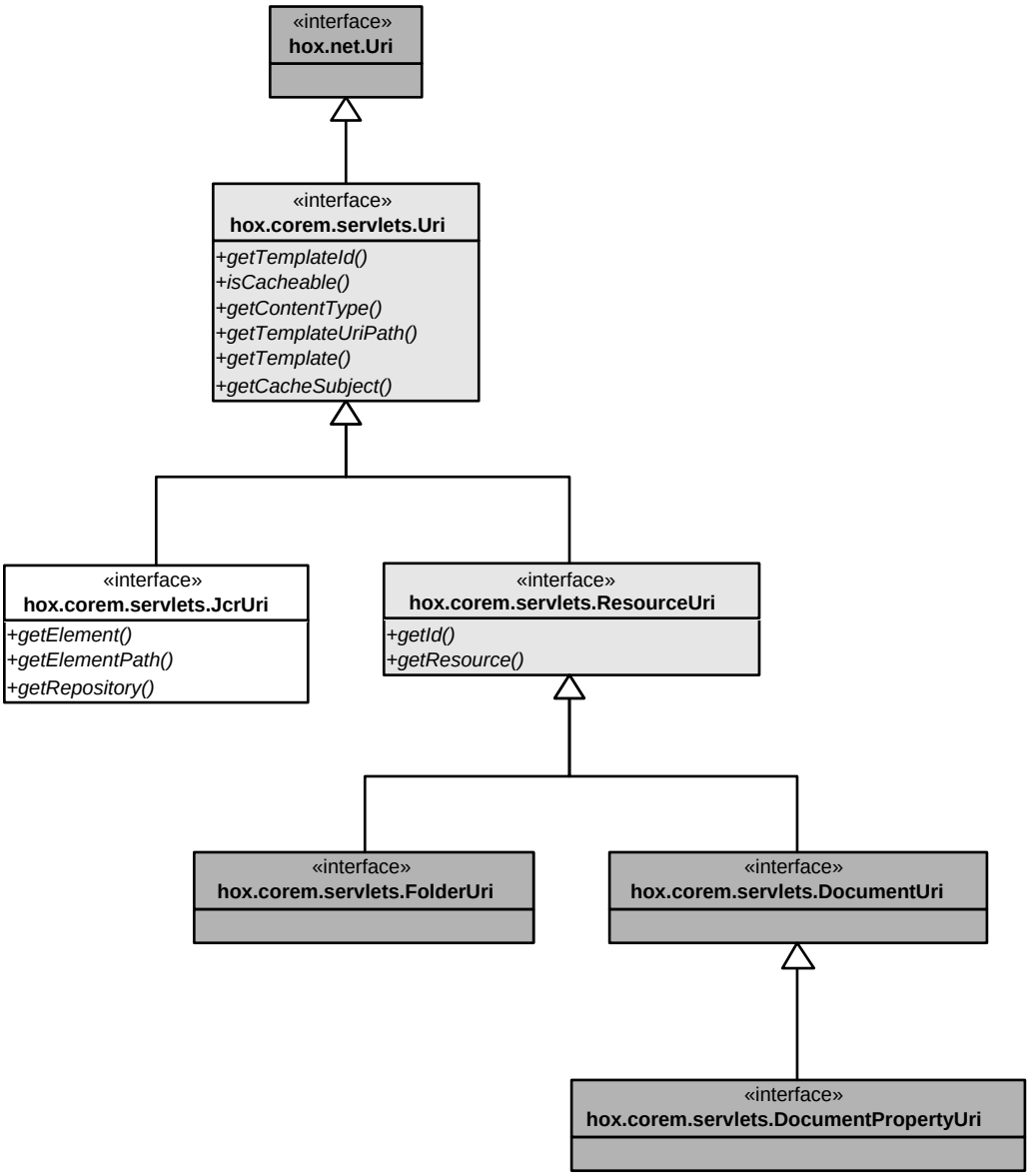

Abbildung 22: Klassendiagramm der Erweiterung der Adressierung von Ressourcen

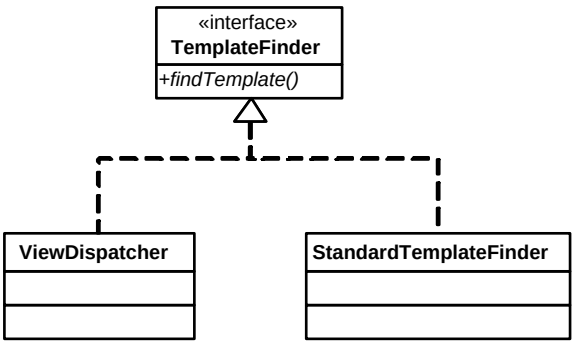

Abbildung 23: Klassendiagramm des Ist-Zustandes des Auffindens von Vorlagen

„objektorientierte“ Sichtweise des Vorlagenmechanismus möglich. Die Vorlagen werden im lokalen Dateisystem gespeichert, was im Folgenden beschrieben wird:

- Zu jedem Dokumenttypen gibt ein entsprechendes Verzeichnis, das Dokumenttypverzeichnis.
- Eine Vorlagendatei in einem Dokumenttypverzeichnis wird als „Methode“ des zugehörigen Dokumenttyps aufgefasst und hier als Ansicht bezeichnet.
- Der `ViewDispatcher` stellt einen `TemplateFinder` dar, welcher „Code-Vererbung“ für Vorlagen ermöglicht. Wenn ein Dokumenttypverzeichnis keine Vorlage mit dem entsprechenden Namen enthält, sucht der `ViewDispatcher` in dem Verzeichnis, das dem generelleren Typs des Dokumenttyps entspricht.
- So ist es möglich, dass ein speziellerer Dokumenttyp eine Ansicht „überschreibt“, indem eine Vorlage mit demselben Namen in das entsprechende Dokumenttypverzeichnis gelegt wird.

Bei der Erweiterung des Prozesses zur Auffindung von Vorlagen besteht zunächst das Problem, dass die `TemplateFinder`-Schnittstelle nur eine Methode zur Auslösung eines Vorlagennamens für Objekte vom Typ `ResourceUri` bietet. Wie in Abschnitt 5.3.1 beschrieben, wurde aus `ResourceUri` eine allgemeinere Schnittstelle `Uri` extrahiert. Diese enthält keine Informationen über die adressierte Ressource und lässt sich so als kleinste gemeinsame Schnittstelle für `Uri`-Objekte, die eine Ressource und eine Vorlage adressieren sollen, auffassen. Deshalb wurde die `TemplateFinder` Schnittstelle etwas generalisiert, indem die `findTemplate`-Methode nun ein Objekt vom Typ `Uri` erwartet und nicht vom zu speziellen Typ `ResourceUri`.

Da die vorhandenen Implementierungen der `TemplateFinder`-Schnittstelle nur mit `ResourceUri`-Objekten umgehen können und es nicht wünschenswert ist, die bestehenden Klassen anzupassen, wird eine weitere Implementierung der `TemplateFinder`-Schnittstelle eingeführt, welche den Aufruf der `findTemplate`-Methode für eine spezielle `Uri`-Klasse an

ein zuvor bekannt gemachtes `TemplateFinder`-Objekt weiterleitet. Da dieses dem *Delegate*-Entwurfsmuster [GHJV95] entspricht, wird die zugehörige Klasse `DelegateTemplateFinder` genannt (siehe Abbildung 24).

Um eine Vorlage für eine `JcrUri` zu finden, wird ein neuer `TemplateFinder` vom Typ `JcrViewDispatcher` implementiert. Da dieser einen großen Teil der Funktionalität und Logik der `ViewDispatcher`-Klasse für `ResourceUri`-Objekte wieder verwenden kann, werden die Methoden, welche dieses implementieren in eine neue Klasse `AbstractViewDispatcher` übertragen und diese, wie Abbildung 24 zeigt, zur Superklasse von `ViewDispatcher` und `JcrViewDispatcher` gemacht.

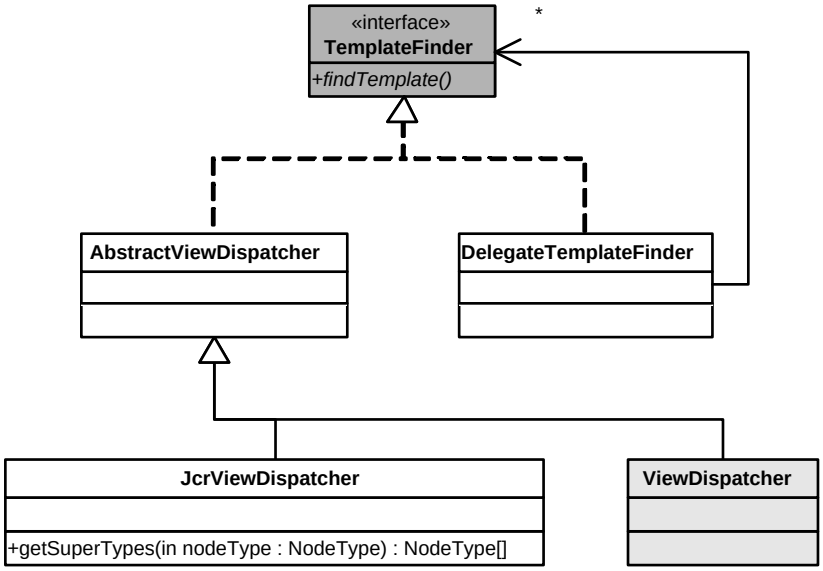

Abbildung 24: Klassendiagramm der Erweiterung des Auffindens von Vorlagen

Die Logik der `ViewDispatcher`-Klasse kann deshalb benutzt werden, weil die Knoten und Elemente in einem *Java Content Repository* und Verzeichnisse und Dokumente im SCT Repository analog in einer Hierarchie angeordnet sind. So kann der Programmcode zum Durchlaufen dieses Baums wieder verwendet werden.

Die Logik zum Durchsuchen des Graphen der Dokumenttypen kann nicht exakt auf die benötige Logik zum Durchlaufen des Graphen der Knotentypen abgebildet werden. Dieses liegt daran, dass der Graph der SCT Dokumenttypen ein Baum und der der *JCR* Knotentypen ein gerichteter, azyklischer Graph ist. Somit stellt letzterer eine komplexere Struktur dar. Der Algorithmus zur Suche einer Vorlage basiert beim `ViewDispatcher` darauf, den Baum der Dokumenttypen solange zur Wurzel hin zu durchlaufen, bis eine passende Vorlage für einen Typen gefunden wurde. Hierbei ergibt sich eine lineare Liste, die durchsucht werden muss da jeder SCT Dokumenttyp höchsten einen Supertypen besitzt. Bei einem *Java Content*

Repository kann ein Knoten aber mehrere Supertypen besitzen. Somit ergibt sich wiederum ein gerichteter, azyklischer Graph, der durchsucht werden muss, um eine geeignete Vorlage zu finden. Da es mehrere Möglichkeiten gibt einen gerichteten, azyklischen Graphen zu durchlaufen, wurde hier eine Breite-zu-erst-Suche implementiert. Dieses geschieht aus dem Grund, dass Typen, die in der Hierarchie näher an dem Ausgangstypen liegen als signifikanter angesehen werden.

Da die Supertypen eines Knotentypen in einem *Java Content Repository* keiner Ordnung unterliegen, ist es nicht voraussehbar, welcher Knoten bei der Breite-zu-erst-Suche ausgewählt wird, falls auf einer Ebene mehrere Knoten existieren. Deshalb implementiert die Klasse `JcrViewDispatcher` das *Abstract Method* Entwurfsmuster [GHJV95]. Die Methode `getSuperTypes` kann überschrieben werden, um für einen gegebenen Knotentyp eine Liste der Supertypen zu liefern, die dann von Anfang bis Ende durchlaufen werden kann, um deterministisch einen Knoten auszuwählen. Die bereits vorhandene Implementierung liefert eine nach dem Knotentypnamen alphabetisch sortierte Liste.

5.3.3 Erzeugung der URIs

In der bestehenden Implementierung, in der es nicht möglich war externe Informationssysteme anzubinden, werden `Uri`-Objekte durch eine Instanz der Klasse `ResourceUriFactory` erzeugt. Diese enthält, entsprechend dem *Factory*-Entwurfsmuster [GHJV95], mehrere Methoden um aus Objekten vom Typ `Resource`, `Uri`-Objekte des Typs `ResourceUri` zu erzeugen. Diesen Ist-Zustand zeigt Abbildung 25.

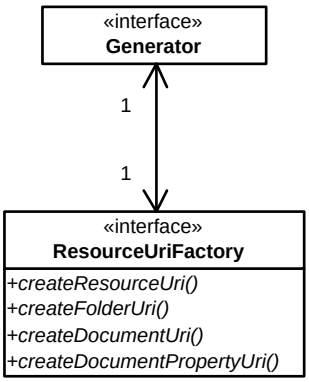

Abbildung 25: Klassendiagramm des Ist-Zustandes der URI-Erzeugung

Um die Erstellung von `Uri`-Objekten zu generalisieren, wird eine neue generelle *Factory*-Schnittstelle vom Typ `UriFactory` eingeführt. Instanzen von Klassen, die diese Schnittstelle implementieren, können für ein gegebenes Objekt ein `Uri`-Objekt erzeugen, sofern die *Factory*-Klasse die Klasse des gegebenen Objektes unterstützt. Damit der *Active Delivery Server* oder, genauer gesagt, dessen Objekt-Repräsentation, das `Generator`-Objekt, die passende *Factory*-

Klasse zur Erzeugung einer `Uri` für ein gegebenes Objekt finden kann, muss jede `UriFactory` über die Methode `canHandle` mitteilen können, ob es für das gegebene Objekt eine `Uri` erstellen kann.

Die Schnittstelle `Resource`, deren Inkarnationen Ressourcen aus dem SCT Repository darstellen, enthält eine Methode `getUri`, die es erlaubt auf eine einfache Weise ein `ResourceUri`-Objekt zu erstellen, welches diese Ressource adressiert.

Da im allgemeinen Fall die Schnittstelle des Objekts, welches adressiert werden soll, keine Methode bietet, um ein `Uri`-Objekt zu erzeugen, muss hierfür eine andere Möglichkeit gefunden werden.

In einer Anwendung, die für den *Active Delivery Server* entwickelt wird, steht zu jedem Zeitpunkt ein Objekt vom Typ `Context` zur Verfügung, welches Informationen über den aktuellen Ausführungskontext besitzt. Weil dieses Objekt immer erreichbar ist und alle notwendigen Information bereitstellt, um an die `UriFactory`-Objekte zu gelangen, wurde hier eine Methode `createUri` eingefügt, die für ein gegebenes Objekt eine `Uri` erstellt, indem sie den Aufruf an die passende `UriFactory` weiterleitet.

Um `JcrUri`-Objekte für Elemente aus einem *Java Content Repository* zu erzeugen, wurde eine Klasse `JcrUriFactory` implementiert.

5.3.4 Formatierung der URIs

Ein Objekt vom Typ `Uri` adressiert ein Objekt im *Active Delivery Server*, welches mit Hilfe einer Vorlage dargestellt werden soll. So ein Objekt kann zum Beispiel eine SCT Ressource oder ein Knoten aus einem *Java Content Repository* sein, welches als HTML-Seite oder als PDF-Dokument dargestellt werden soll.

Um mit einem Internet Browser eine Seite aufzurufen, gibt man eine URI als Zeichenkette ein. Auch innerhalb einer HTML-Seite muss eine Referenz auf eine andere Seite aus dem *Active Delivery Server* durch eine Zeichenkette dargestellt werden.

Es gibt also zwei Darstellungsformen von URIs. Einmal als `Uri`-Objekt in der objektorientierten Programmiersprache und einmal als Zeichenkette, zum Beispiel als Referenz in einer HTML-Datei. Diese Darstellungen sind äquivalent und können daher ineinander umgewandelt werden. Es existiert also eine bijektive Abbildung zwischen `Uri`-Objekt und Zeichenketten-Repräsentation.

Die Umwandlung eines `Uri`-Objektes in eine Zeichenkette wird URI-Formatierung oder kurz Formatierung genannt. Der umgekehrte Weg, die Erzeugung eines `Uri`-Objektes aus einer Zeichenkette wird URI-*Parsing* oder kurz *Parsing* genannt.

Da es in der bisherigen Implementierung nur `ResourceUri`-Objekte gibt um URIs für SCT Ressourcen zu repräsentieren, gibt es eine Klasse `ResourceUriFormat`, deren Subklassen für die Konvertierung zwischen Objekten vom Typ `ResourceUri` und diversen Zeichenkettendarstellungen zuständig sind. Es existiert eine generelle Schnittstelle vom Typ `UriFormat`, deren Instanzen allgemeine URIs konvertieren können.

Für die Formatierung eines `JcrUri`-Objektes zu einer Zeichenkette bzw. die Analyse einer Zeichenkette, um daraus ein `JcrUri`-Objekt zu erstellen, wird die Schnittstelle `JcrUriFormat`

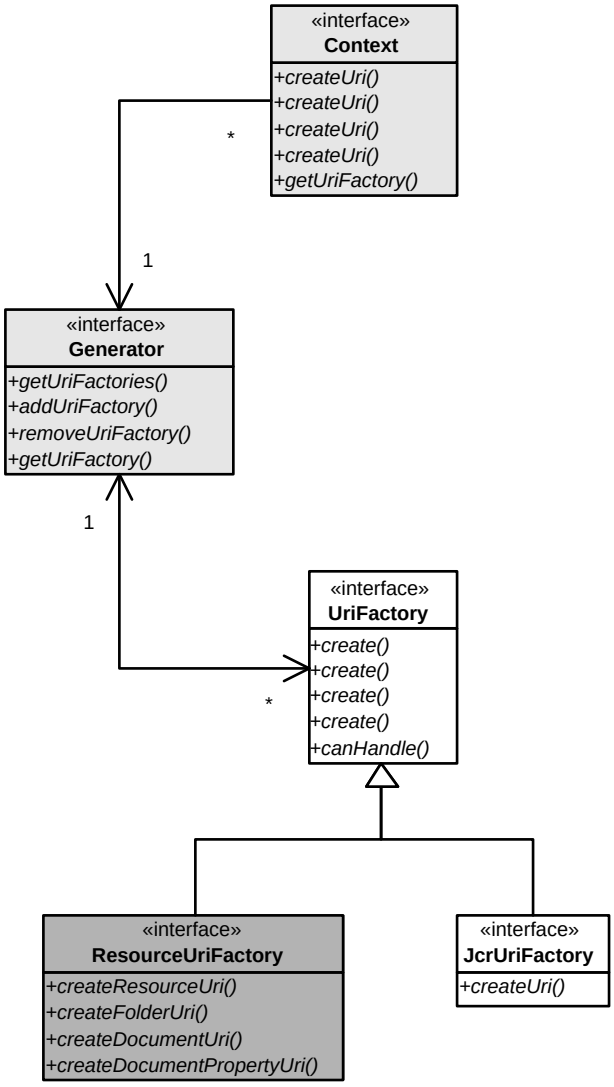

Abbildung 26: Klassendiagramm der Erweiterung der URI-Erzeugung

eingeführt. Implementierungen dieser Schnittstelle vom Typ `StandardJcrUriFormat` erzeugen bzw. verstehen Zeichenketten ähnlich zu denen, die Objekte der bestehenden Klasse `StandardResourceUriFormat` erzeugen.

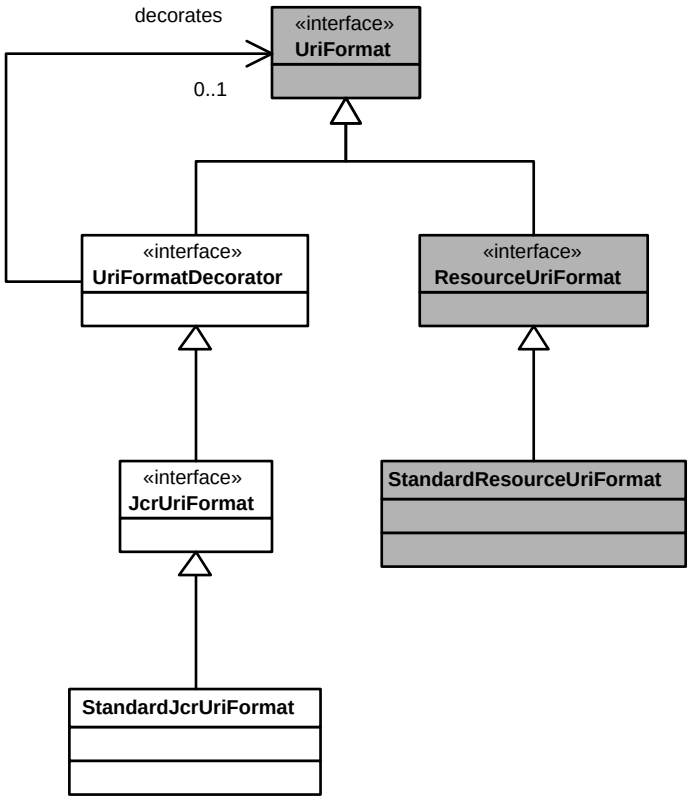

Abbildung 27: Klassendiagramm der Erweiterung der URI-Formatierung

Da ein Objekt vom Typ `UriFormat` in der Regel nur eine bestimmte Klasse von `Uri`-Objekten formatieren bzw. parsen kann, muss eine Möglichkeit gefunden werden, ein konkretes `Uri`-Objekt an das `UriFormat` weiterzuleiten, welches damit umzugehen weiß. Dazu wurde das *Decorator* Entwurfsmuster aus [GHJV95] gewählt. Wie Abbildung 27 zeigt, erweitert die Schnittstelle `UriFormatDecorator` die Schnittstelle `UriFormat` und hält eine Referenz auf das zu dekorierende `UriFormat`. Die Klasse `StandardJcrUriFormat` stellt einen konkreten *Decorator* dar, der zum Beispiel genutzt werden kann, um das `StandardResourceUriFormat` zu dekorieren. Konkrete *Decorator* können entscheiden, ob sie aus einem gegeben `Uri`-Objekt eine Zeichenkette erzeugen können oder ob sie das Objekt an das dekorierte `UriFormat` weiterleiten. Objekte vom Typ `ResourceUriFormat` können nur am Ende einer *Decorator*-Kette stehen, da sie selbst aus Kompatibilitätsgründen kein *Decorator* sind.

5.3.5 Verfolgung der Abhängigkeiten

Eines der Hauptmerkmale des *Active Delivery Servers* ist die Verfolgung von Abhängigkeiten im Zusammenhang mit der Zwischenspeicherung der generierten Seiten (HTML-Seiten, XML-Fragmente, JPEG-Bilder, etc.). Eine generierte Seite, zum Beispiel eine HTML-Seite, basiert auf mehreren Elementen, wie der verwendeten Vorlage, den angezeigten SCT Ressourcen und HTML-Fragmenten, welche in die Seite eingefügt werden. HTML-Fragmente basieren ihrerseits wieder auf einer Vorlage und anderen SCT Ressourcen und womöglich wiederum anderen HTML-Fragmenten. Somit hängt eine generierte HTML-Seite von vielen Dingen ab, welche die endgültige Erscheinung der Seite beeinflussen. Diese Abhängigkeiten ergeben einen gerichteten, azyklischen Graphen, der hier Abhängigkeitsgraph genannt wird. Die Kanten in diesem Graphen zeigen von einem abhängigen Objekt (*dependant*) zu dessen Abhängigkeiten (*dependencies*).

Im Objektmodell des *Active Delivery Servers* sieht das Konzept der Abhängigkeiten wie in Abbildung 28 gezeigt aus. Das *Model-View*-Konzept wurde um das Konzept der Abhängigkeiten erweitert. Die Schnittstelle `Dependency` erweitert das Konzept der `Model`-Schnittstelle. Durch diese Spezialisierung kann eine Ansicht, also ein Objekt vom Typ `View` bzw. `DependentModel`, von mehreren Objekten vom Typ `Model` bzw. `Dependency` abhängig gemacht werden. Dadurch entsteht der oben genannte Abhängigkeitsgraph.

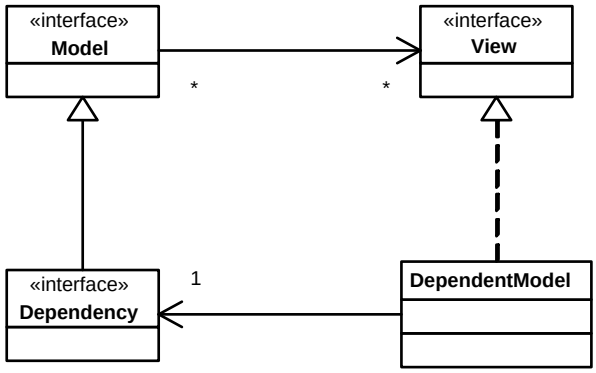

Abbildung 28: Konzept der Abhängigkeiten

Im Beispiel der Auslieferung von HTML-Seiten sind die erzeugten HTML-Dateien und -Fragmente Ansichten, also vom Typ `DependentModel`. Diese sind von SCT Ressourcen und Vorlagen-Dateien abhängig, für welche `Dependency`-Objekte erzeugt werden.

Um nun während der Generierung einer Seite immer die korrekten Abhängigkeiten zu erzeugen, existiert ein Objekt vom Typ `DependencyTracker`. Dieses verwaltet auf einem Stapel die aktuell generierte Ansicht, den so genannten `Trackee`. Diesem werden beim Zugriff auf die Programmierschnittstelle des *Active Delivery Servers* die notwendigen Anhängigkeiten zugeordnet. Diesen Zusammenhang stellt Abbildung 29 dar.

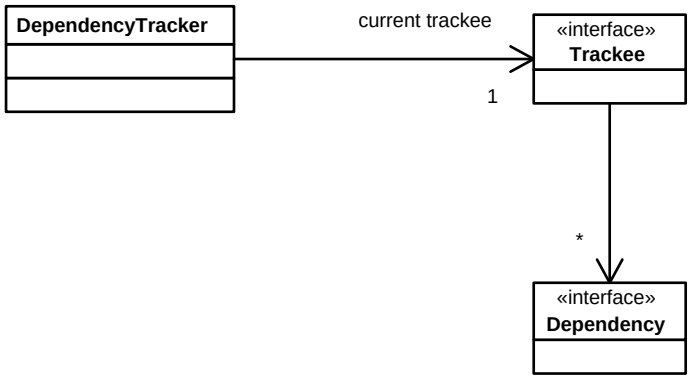

Abbildung 29: Konzept der Verfolgung der Abhängigkeiten

Da ein Aufruf des `DependencyTracker`-Objekts notwendig ist, um die Abhängigkeit von einem benutzten Objekt zu einem verfolgten Objekt zu erzeugen, ergeben sich folgende Möglichkeiten dieses zu implementieren:

- **Manueller Aufruf**. Es ist möglich, die Abhängigkeiten manuell zu erzeugen. Dieses kann zum Beispiel im Quelltext einer Vorlage geschehen, indem für jedes Objekt, welches in die Ausgabe einbezogen wird, ein Aufruf an den `DependencyTracker` getätigt wird. Trotz des Vorteils der feingranularen Kontrolle über die Erzeugung von Abhängigkeiten überwiegt hier jedoch der Nachteil, dass Objekte, die für die Generierung der Ausgabe benötigt werden, übersehen werden können und so keine Abhängigkeiten zwischen diesen Objekten erzeugt werden. Dies kann zum Beispiel dadurch geschehen, dass ein Objekt ein anderes referenziert und von dessen Zustand abhängt. Dieser Zusammenhang kann aber durch die Schnittstelle des Objektes verborgen werden und somit unentdeckt bleiben. Somit kann die Ausgabe einer Vorlage im Zwischenspeicher bereits veraltet sein, ohne das dieses durch die ereignisbasierte Invalidierung behoben wird, da keine Abhängigkeit zum veränderten Objekt besteht.

- **Erweiterung der Datenzugriffsschicht**. Besteht die Möglichkeit die Datenzugriffsschicht direkt zu ändern, können die Aufrufe an den `DependencyTracker` dort eingebaut werden. Dieses ist natürlich nur machbar, sofern diese Schicht nicht als Bibliothek angebunden wird und somit kein Eingriff in die Quelltexte möglich ist. Die Programmierschnittstelle des *Active Delivery Servers* wurde so realisiert.

- *Wrapper*-**Schicht**. Die Aufrufe des `DependencyTracker` finden in einer *Wrapper*-Schicht statt, welche die Bibliothek, die zum Datenzugriff verwendet wird, kapselt. In dieser Schicht wäre es dann möglich zu einem gegebenen Aufruf alle notwendigen Abhängigkeiten zu erzeugen. Ist die Schnittstelle der Bibliothek jedoch sehr umfangreich, ist es erforderlich sehr viele *Wrapper*-Klassen zu erstellen. Ändert sich die Schnittstelle,

wie zum Beispiel bei einem Versionssprung, muss die *Wrapper*-Schicht ebenfalls überarbeitet werden.

- **Dynamische Stellvertreter-Klassen**. Die Objekte, welche die Programmierschnittstelle zu Datenzugriff bietet, werden durch Objekte von dynamischen Stellvertreter-Klassen gekapselt, welche die Funktionalität zur Erzeugung von Abhängigkeiten zu der verwendeten Bibliothek hinzufügen. Diese Option wurde für die Erzeugung der Abhängigkeiten von *Java Content Repository* Elementen gewählt. Die anderen vorgestellten Optionen wurden aus den genannten Nachteilen verworfen. Im Folgenden wird diese Lösung näher erläutert.

Dynamische Stellvertreter-Klassen

In objektorientierten Programmiersprachen, wie der Sprache Java [GJSB00], können Klassen eine oder mehrere Schnittstellen implementieren. Die Angabe der Schnittstellen, die diese Klasse implementieren soll, geschieht in der Regel im Quelltext der Klasse in einer von der verwendeten Sprache abhängigen Klausel in der Klassendefinition. Damit stehen die Schnittstellen zur Kompilierzeit fest und können zur Laufzeit des Programms nicht geändert werden.

In der Java 2 Plattform wurden die dynamischen Stellvertreter-Klassen, auch dynamische Proxy-Klassen genannt, eingeführt. Diese erlauben es eine Klasse zu erstellen, deren Schnittstellen zur Laufzeit des Programms festgelegt werden können. Ein Methodenaufruf an eine solche Klasse durch eine der Schnittstellen wird dann an ein anderes Objekt durch eine einheitliche Schnittstelle weitergeleitet. So kann ein typsicheres Stellvertreter-Objekt für eine Reihe von Schnittstellen erzeugt werden, ohne die Hilfe von Kompilierzeitwerkzeugen wie Code-Generatoren zu benötigen.

Jede Instanz einer Stellvertreter-Klasse, Proxy-Instanz genannt, referenziert ein Objekt, welches Methodenaufrufe entgegennehmen und verarbeiten kann. Dieses Objekt ist vom Typ `InvocationHandler` (siehe Abbildung 30). Die `InvocationHandler`-Schnittstelle besitzt genau eine Methode, an welche der Aufruf kodiert weitergeleitet wird. Ein Methodenaufruf an eine Instanz einer dynamischen Proxy-Klasse wird also an die einzige Methode des `InvocationHandler`-Objekts der Proxy-Instanz weitergeleitet. Die aufgerufene Methode wird dabei als `Method`-Objekt und die Parameter als ein Feld vom Typ `Object` kodiert.

Dynamische Proxy-Klassen sind ein Teil der *Reflection*-API der Java 2 Plattform.

Um innerhalb des *Active Delivery Servers* die Abhängigkeiten von einem *Java Content Repository* zu verfolgen, wird die Programmierschnittstelle mit dynamischen Stellvertreter-Objekten gekapselt. Jedes Objekt, welches durch die *Java Content Repository*-API geliefert wird, wird durch ein Proxy-Objekt verdeckt. Der `InvocationHandler` des Proxy-Objekts erzeugt bei einem Aufruf des Stellvertreters eine Abhängigkeit mit Hilfe des `DependencyTrackers` und leitet den Aufruf an das gekapselte Objekt weiter. Liefert dieser Aufruf ein JCR-Objekt zurück, muss dieses wiederum gekapselt werden. Wird ein verdecktes JCR-Objekt als Parameter in einem Aufruf an ein Stellvertreter-Objekt übergeben, so muss diese enthüllt werden, bevor es an die JCR API-Methode übergeben wird. So werden

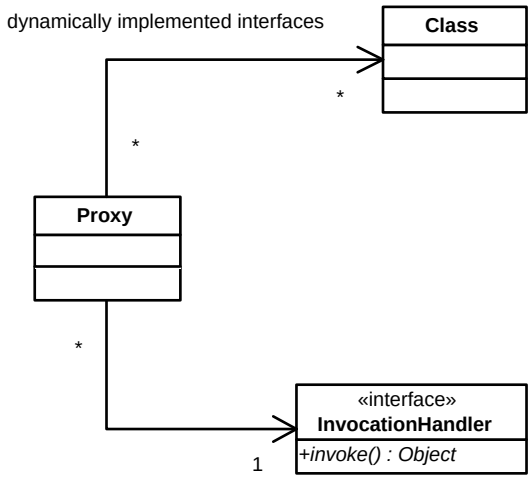

Abbildung 30: Dynamische Stellvertreterobjekte

Inkompatibilitäten der *Java Content Repository* Implementierung mit den Stellvertreter-Objekten vermieden. Insgesamt ist es so mit wenig Aufwand möglich, trotz der komplexen Schnittstelle der JCR API, Abhängigkeiten zu verfolgen.

5.3.6 Ereignisbasierte Invalidierung

Damit der *Active Delivery Server* immer einen aktuellen Stand der generierten und zwischengespeicherten Seiten ausliefern kann, müssen diese im Zwischenspeicher als ungültig (*invalid*) markiert werden, sobald sich die Objekte, von denen sie abhängen, geändert haben.

Hierfür ist die ereignisbasierte Invalidierung zuständig. Sie ist ebenfalls ein Teil des *Active Delivery Servers*. Dieser empfängt Ereignisse über die Änderung von Objekten und invalidiert die Objekte im Zwischenspeicher, die von dieser Änderung betroffen sind. Um die passenden Objekte im Zwischenspeicher zu finden, wird der Abhängigkeitsgraph ausgehend von dem Objekt, für welches das Ereignis empfangen wurde, bis zu den Blättern durchlaufen. Alle Knoten die dabei berührt werden, werden als ungültig markiert.

Eine Implementierung der Übereinstimmungsebene 2 der *Java Content Repository* Spezifikation muss „Beobachtung“ (*Observation*) unterstützen. Deshalb ist eine ereignisbasierte Invalidierung nur für Implementierungen der Ebene 2 möglich. Dazu macht der *Active Delivery Server* dem Repository ein Objekt vom Typ `EventListener` bekannt, welchem dann Ereignisse über Änderungen am Repository gesendet werden.

Da Implementierungen der Übereinstimmungsebene 1 keine Beobachtung unterstützen, dürfen Vorlagen, die Elemente aus Implementierungen der Ebene 1 benutzten entweder nicht für eine Zwischenspeicherung markiert werden oder die Invalidierung muss auf anderem Wege geschehen.

6 Zusammenfassung und Ausblick

Ausgehend von dem Problem der Kommunikation von n Klientensystem mit m Informationssystemen und den dabei vorhandenen $n \times m$ Schnittstellen (siehe Kapitel 1) wurde untersucht, in wie weit sich eine einheitliche Schnittstelle finden lässt, die häufig auftretende und verallgemeinerbare Dienste von Informationssystemen umfasst. Es wurden Dienste wie Datenbanktechnologie, Transaktionsverarbeitung, Versionsverwaltung und Überwachung genauer betrachtet. Diese und weitere Dienste werden unter dem Begriff Repository-Technologie zusammengefasst. Es wurden einige vorhandene Schnittstellen auf die Anforderungen der eben genannten Dienste untersucht und schließlich die *Content Repository API for Java Technology* [JCR03] als geeignete Schnittstelle ausgewählt und einer genaueren Betrachtung unterzogen (siehe Kapitel 2 und 3). Als Anwendungsbeispiel wurde die gewählte Schnittstelle in ein vorhandenes Content Management System integriert (siehe Kapitel 4 und 5).

Während in der Datenbanktechnologie die vorhandenen Dienste wie Anfragesprachen, Datendefinition und -manipulation sowie Transaktionsverwaltung durch Standards wie SQL [ISO92], XA [XA92] und ODMG [CBB+00] definiert sind und sich durch viele am Markt vorhandene Produkte als einsatzfähig bewiesen haben, fehlen entsprechende Standards für die Dienste, um welche die Repository-Technologie die Datenbankmanagementsysteme erweitert. Es wurden viele Dokumenten und Content Management Systeme entwickelt, welche Teile der Repository-Technologie unterstützen, aber es hat sich kein einheitlicher Standard herausgebildet.

Die *Content Repository API for Java Technology* (JCR) ist ein erster Schritt zur Standardisierung von Diensten der Repository-Technologie. Die Ausarbeitung einer Spezifikation schreitet jedoch nur langsam voran. Nach dem der angestrebte Termin im Juli 2003 für eine endgültige Version der Spezifikation nicht gehalten werden konnte, wurde er um ein Jahr verschoben. Ob dieser Termin haltbar ist, darf angezweifelt werden, da einige Meilensteine auf dem Weg dorthin noch nicht erreicht wurden. Über einige wichtige offene Punkte, wie Versionsverwaltung und Anfragesprachen, scheint noch Uneinigkeit zu herrschen und eine Lösung dafür ist noch nicht in Sicht. Nichts desto trotz enthält JCR aber einige interessante Gesichtspunkte. Der Entwurf der Schnittstelle zum Zugriff auf Repository-Objekte sowie die Semantik zur Manipulation dieser Objekte erscheinen gut gelungen.

Weiterhin steht und fällt die Akzeptanz der *Content Repository API for Java Technology* mit der Unterstützung der Hersteller von Content Management Systemen und der Open Source Gemeinde. Zur Zeit scheint JCR nur von der Day Software AG und kleineren Firmen in deren Umfeld vorangetrieben zu werden. Aussagen weiterer Firmen zu JCR fehlen im Moment. Auch der Rückzug der Referenzimplementation aus dem Jakarta Slide Projekt wirft kein gutes Licht auf die Zukunft von JCR.

Die JCR Schnittstelle in die *CoreMedia Smart Content Technology* (SCT) zu integrieren erwies sich als schwierig, da die vorhandenen Komponenten nicht besonders gut darauf vorbereitet waren externe Ressourcen (aus der Sicht von SCT) einzubinden. Daher mussten die betroffenen Komponenten bis in die Tiefe hinein erweitert werden. Dazu wurden Schnittstellen in allgemeine und SCT-spezifische aufgeteilt, um anhand der allgemeinen Schnittstellen die

Integration von JCR vorzunehmen. Da die JCR-Spezifikation noch nicht ihr finales Stadium erreicht hat, wird diese Erweiterung ein Prototyp bleiben. Es hat sich gezeigt, dass die Änderungen an den vorläufigen Versionen der Spezifikation teilweise zu gravierend waren, um die Implementierung auf eine stabile Basis stellen zu können.

Die Repository-Technologie stellt eine interessante Erweiterung der Datenbank-Technologie dar. Viele der Funktionen und Dienste, wie sie heute in Dokumenten und Content Management Systemen für jedes Produkt neu implementiert werden, könnten auf Dauer in Repository-Produkte integriert werden. Dieses ist analog zu der Entwicklung bei den Datenbanken zu sehen, wo heutzutage die Datenbankfunktionalität nicht für jedes Informationssystem neu entwickelt wird, sondern auf ein Datenbankprodukt zurückgegriffen wird.

A JCRQL Syntaxdefinition

Dieser Anhang beschreibt die Syntax der JCRQL-Anfragesprache (siehe Abschnitt 3.2.3). Als syntaktische Metasprache wird hier die erweiterte Backus-Naur-Form [ISO96b] verwendet. Um diese Definition einfacher und lesbarer zu gestalten wurden einige Definitionen durch Kommentare ersetzt.

```
query = selectclause, fromclause, [locationclause], [whereclause],
        [textsearchclause], [orderclause];

selectclause = "SELECT", ("*" | nodenamelist);

nodenamelist = nodename, {",", nodename};

nodename = (* The name of a node *);

fromclause = "FROM", ("*" | nodetypelist);

nodetypelist = nodetype, {",", nodetype};

nodetype = (* The name of a node type *);

locationclause = "LOCATION", [depthclause], [followingclause], pathpattern;

pathpattern = plainpath,["/"] | plainpath, ["//", [plainpath, ["/"]]];

plainpath = itempattern, {"/", itempattern};

itempattern = "*" | ["*"], fragment, {fragment};

fragment = char | char*;

char = (* Any character valid within an item name *);

depthclause = "DEPTH", number;

number = (* An integer *);

followingclause = "FOLLOWING", ["CHILDREN"], softlinkprops;

softlinkprops = itempattern, {",", itempattern};

whereclause = "WHERE", expression;
```

```
expression = property, op, value |
             property, "LIKE", pattern |
             expression, "AND", expression |
             expression, "OR", expression |
             "NOT", expression |
             "(", expression, ")";

op = "=" | ">" | "<" | ">=" | "<=";

property = (* The name of a property *);

value = (* A property value (in standard string form) *);

pattern = '"', likepattern, '"' | likepattern;

likepattern = (wildcard | [wildcard], likefragment, {likefragment});

likefragment = likechar | likechar, wildcard;

wildcard = "*" | "?" | "%" | "_";

char = (* Any character valid within the string representation of a value
          (except for the escaped characters) *);

textsearchclause = "TEXTSEARCH", searchexp;

searchexp = simplesearchexp |
            customsearchexp |
            '"', simplesearchexp, '"' |
            '"', customsearchexp, '"';

simplesearchexp = ["-"], term {["OR"], ["-"], term};

term = word | "'", word, {word}, "'";

word = (* A string containing no whitespace
          (the special characters) *);

customsearchexp = (* A search expression specified by an external standard *);

orderclause = "ORDER BY", property, {",", property};
```

Literatur

[ABGS91] R. Agrawal, S. Buroff, N. H. Gehani, and D. Shasha. Object versioning in Ode. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 446–455, Kobe, Japan, 1991.

[BCF+03] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, November 2003. http://www.w3.org/TR/xquery.

[BD94] Philip A. Bernstein and Umeshwar Dayal. An Overview of Repository Technology. In *Proceedings of the 20th VLDB Conference*, pages 705–713, 1994.

[Ber97] Philip A. Bernstein. Repositories and Object Oriented Databases. In Klaus R. Dittrich and Andreas Geppert, editors, *Proceedings BTW '97*, pages 34–46, Ulm, Germany, 1997. Springer-Verlag.

[BHS+97] Philip A. Bernstein, Brian Harry, Paul Sanders, David Shutt, and Jason Zander. The Microsoft Repository. In *Proceedings of International Conference on Very Large Data Bases (VLDB '97)*, pages 3–12, Athens, Greece, 1997.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL), 1994. RFC 1738.

[BM01] Paul V. Biron and Ashok Malhotra. *XML Schema Part 0: Datatypes*. World Wide Web Consortium, Mai 2001. http://www.w3.org/TR/xmlschema0.

[BN97] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium, Oktober 2000. http://www.w3.org/TR/REC-xml.

[BSL00] Don Box, Aaron Skonnard, and John Lam, editors. *Essential XML - Beyond Markup*. DevelopMentor Series. Addison-Wesley, September 2000.

[CAE+02] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning Extensions to WebDAV, März 2002. RFC 3253.

[CBB+00] R. Cattell, Douglas Barry, Mark Berler, Jeff Eastmann, Jordan, Craig Russell adn Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, April 2000.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium, November 1999. http://www.w3.org/TR/xpath.

[Cla99] James Clark. *XSL Transformations (XSLT) Version 1.0*. World Wide Web Consortium, November 1999. http://www.w3.org/TR/xslt.

[CM03a] CoreMedia AG. *Smart Content Infrastructure Administration and Operation Manual 4.1.36*, 2003.

[CM03b] CoreMedia AG. *Smart Content Infrastructure Developer Manual 4.1.36*, 2003.

[CM03c] CoreMedia AG. *Smart Content Infrastructure Workflow Developer Manual 4.1.36*, 2003.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[COR02] Common Object Request Broker Architekture: Core Specification, Dezember 2002. http://www.omg.org/.

[Cow01] Danny Coward. *Java Servlet Specification Version 2.3*. Sun Microsystems, Inc., 2001.

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Lecture Notes in Computer Science*. Springer-Verlag, Dezember 2000. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf.

[CT01] John Cowan and Richard Tobin. *XML Information Set*. World Wide Web Consortium, Oktober 2001. http://www.w3.org/TR/xml-infoset.

[CW97] Reidar Conradi and Bernhard Westfechtel. Towards a uniform version model for software configuration management. In *System Configuration Management*, pages 1–17, 1997.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[DCE97] The Open Group. *CAE Specification DCE 1.1: Remote Procedure Call*, 1997. http://www.opengroup.org/onlinepubs/009629399/.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML, 1998. http://www.research.att.com/ mff/files/final.html.

[EH01] Jon Ellis and Linda Ho. *JDBC 3.0 Specification*. Sun Microsystems Inc., Oktober 2001.

[Fal01] David C. Fallcide. *XML Schema Part 0: Primer*. World Wide Web Consortium, Mai 2001. http://www.w3.org/TR/xmlschema-0.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. RFC 2616.

[FMM+03] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Jonathan Marsh, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. World Wide Web Consortium, November 2003. http://www.w3.org/TR/xpath-datamodel.

[Gei95] Kyle Geiger. *Inside ODBC*. Microsoft Press, August 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2000.

[GMMW03] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. *XPointer Framework*. World Wide Web Consortium, März 2003. http://www.w3.org/TR/xptr-framework/.

[Gri04] Richard Grimes. Revolutionary file storage system lets users search and manage files based on content. *MSDN Magazine*, Januar 2004.

[Ham96] Graham Hamilton. *JavaBeans*. Sun Microsystems, Inc., 1996.

[HBS+02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service*. Sun Microsystems, Inc., April 2002.

[HHW+03] Arnaud Le Hors, Phillipe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 3 Core Specification*. World Wide Web Consortium, 2003. http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107/.

[ISO92] International Standards Organization / International Electrotechnical Commission. *ISO/IEC 9075:1992 Database Language SQL*, 1992.

[ISO96a] International Standards Organization / International Electrotechnical Commission. *ISO/IEC 14834:1996, Information technology - Distributed transaction processing - The XA specification*, 1996.

[ISO96b] International Standards Organization / International Electrotechnical Commission. *ISO/IEC 14977:1996(E) Extended BNF*, 1996.

[ISO98] International Standards Organization / International Electrotechnical Commission. *ISO/IEC 10026, Information technology - Open systems interconnection - Distributed transaction processing*, 1998.

[JCR03] *Content Repository API for Java Technology Specification*, 2003. http://www.jcp.org/en/jsr/detail?id=170.

[JTA99] Sun Microsystems Inc. *Java Transaction API (JTA)*, 1999. http://java.sun.com/products/jta/.

[Kat90] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.

[KCB86] Randy H. Katz, Ellis Chang, and Rajiv Bhateja. Version modeling concepts for computer-aided design databases. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 379–386. ACM Press, 1986.

[MB] Chaoying Ma and Jean Bacon. COBEA: A CORBA-based event architecture. pages 117–132.

[Obj01] Object Management Group. *Event Service Specification*, März 2001. http://www.omg.org/docs/formal/01-03-01.pdf.

[OTS03] Object Management Group. *Transaction Service Specification*, September 2003. http://www.omg.org/cgi-bin/doc?formal/2003-09-02.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL), 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[RPL03] Mark Roth and Eduardo Pelegrí-Llopart. *JavaServer Pages Specification*. Sun Microsystems, Inc., 2003.

[Ses99] Govind Seshadri. Understanding JavaServer Pages Model 2 Architecture. *JavaWorld*, Dezember 1999. http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html.

[Sha01] Bill Shannon. *Java 2 Platform Enterprise Edition Specification, v1.3*. Sun Microsystems, Inc., Juli 2001.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. World Wide Web Consortium, Mai 2001. http://www.w3.org/TR/xmlschema-1.

[UML03] Object Management Group. *OMG Unified Modelling Language Specification*, März 2003. http://www.omg.org/docs/formal/03-03-01.pdf.

[Weg02] Holm Wegner. *Analyse und objektorientierter Entwurf eines integrierten Portalsystems für das Wissensmanagement*. Dissertation, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Januar 2002.

[Whi98] Jim Whitehead. Collaborative Authoring on the Web: Introducing WebDAV. *Bulletin of the American Society for Information Science*, Oktober/November 1998.

[Wie97] Gio Wiederhold. Mediators in the architecture of future information systems. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 185–196. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[WVC03] *WVCM: The Workspace Versioning and Configuration Management API*, 2003. http://www.jcp.org/en/jsr/detail?id=147.

[WW98] Jim Whitehead and Meredith Wiggins. WEBDAV: IETF Standard for Collaborative Authoring on the Web. *IEEE Internet Computing*, September/Oktober 1998.

[XA92] The Open Group. *Distributed TP: The XA Specification*, Februar 1992. http://www.opengroup.org/public/pubs/catalog/c193.htm.