

– Diplomarbeit –

**Analyse und Vergleich des MyCoRe „Content Repository“
Rahmenwerks mit dem Java Content Repository Standard
JSR-170 im Kontext einer offenen und erweiterbaren
Repository-Schnittstelle**

Bearbeiter:

Thomas Scheffler

Betreuer:

Dipl.-Phys. Cataldo Mega, STSM



IBM Deutschland Entwicklung GmbH
ECM Solutions Design and Performance
Schönaicher Straße 220
71032 Böblingen



seit 1558

Prof. Dr. Klaus Küspert

Friedrich-Schiller-Universität Jena
Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
Ernst-Abbe-Platz 2
07743 Jena

Jena, den 29.07.2005

Urheberrechtshinweis

Diese Diplomarbeit ist urheberrechtlich geschützt. Das Copyright der Arbeit liegt bei dem Autor. Alle Rechte, auch Übersetzungen, sind vorbehalten. Die Vervielfältigung und Weitergabe zu privaten, nichtkommerziellen Zwecken ist ausdrücklich gestattet, vorausgesetzt, der Inhalt wird nicht verändert und der Copyrightvermerk bleibt ebenfalls erhalten. Ebenso erlaubt ist die urheberrechtskonforme Verwendung für Zwecke der Forschung und Lehre. Jede anderweitige Nutzung erfordert das vorherige, schriftliche Einverständnis des Autors.

Wettbewerbsrechtlicher Hinweis

Die bloße Nennung von Namen und Produkten von Herstellern, Dienstleistungsunternehmen und Firmennamen dient lediglich als Information und stellt keine Verwendung des Warenzeichens sowie keine Empfehlung des Produktes oder der Firma dar. Daher werden auch für die Verwendung und Nutzung solcher Produkte, Dienstleistungen und Firmen keine Gewähr übernommen.

Haftung für Verweise

Alle in dieser Arbeit angegebenen Verweise (Links) sind rein informativ. Der Autor ist nicht für die Inhalte der verlinkten Seiten verantwortlich.

Kurzfassung

Das Ziel der Open-Source-Initiative MyCoRe (<http://www.mycore.org>) ist die Entwicklung eines herstellerunabhängigen Rahmenwerks für multimediale Online-Informations- und -Bibliothekssysteme. Kernaufgabe dieses „Digital Library Framework“ ist es, die Bedürfnisse des universitären Betriebs in Punkto Unterstützung von Online-Kursentwicklung und Kursangebot, Online Publishing und Dokumentenarchivierung zu befriedigen.

In der vorliegenden Arbeit wird unter anderem evaluiert, inwieweit das MyCoRe-Framework tatsächlich eine offene, erweiterbare und unabhängige Persistenzschicht darstellt und nicht ein Filter geworden ist, der alle repository-spezifischen Archivdienste generalisiert und auf den kleinsten gemeinsamen Nenner reduziert oder gar wegfiltert.

Auf der Basis der erlangten Erkenntnisse werden Entwurfsvorschläge erarbeitet, die aufzeigen, wie man Erweiterbarkeit und die einfache Anbindung von herstellerspezifischen Archivsystemen, wie dem IBM Content Manager, mit ihren Diensten und deren mögliche Vorzüge ausnutzen kann. Dies soll im Sinne einer echten „neutralen integrativen Persistenzschicht“, eventuell mit einer möglichen Adaptierung und Integration der JSR-170 Schnittstelle, geschehen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich genannten Quellen und Hilfsmittel verwendet. Wörtlich oder sinngemäß übernommenes Gedankengut wurde als solches kenntlich gemacht.

Thomas Scheffler
Jena, den 29. Juli 2005

Vorwort

Danksagung

Ich möchte mich an dieser Stelle natürlich zuerst bei meinen Betreuern bedanken. Cataldo Mega ermöglichte mir mit dieser Arbeit nicht nur einen Einblick in das IBM Labor in Böblingen, sondern auch in sein fundiertes Wissen. Seine Fragen waren stets eine „Challenge“ und ich hoffe er ist erfreut zu sehen, dass die Arbeit „just in time“ fertig wurde. Professor Küspert, mein Betreuer in Jena, hat mich bereits sehr gut durch meine Studienarbeit begleitet. Es tut mir Leid, dass so viele rohe Entwürfe dieser Arbeit zu erst an ihn herangekommen sind. Ohne ihn wäre die Arbeit in der Form und vor allem in der (Regel-)Zeit nicht fertig geworden.

Einen speziellen Dank richte ich an Dr. Detlev Degenhart (Universität Freiburg), der mich mit äußerst professionellen Revisionen herausgefordert hat und mich in der schwierigen Anfangsphase besonders unterstützt hat. Den gleichen Dank kann ich so auch an Kathleen Krebs (Universität Hamburg, IBM Labor Böblingen) weiterreichen. Es bleibt mir ein Rätsel, mit welchem Engagement sie meine Arbeit begleitete. Ich hoffe, dass auch ihnen diese Arbeit gerecht wird. Durch beide ist mein Wissen bezüglich UML während dieser Arbeit geformt worden.

Für die finanzielle Unterstützung und Freiräume während der Arbeitszeit möchte ich mich zudem bei Michael Lörzer (Thüringer Universitäts- und Landesbibliothek) bedanken. Er und meine Kollegen in der ThULB, besonders Ulrike Krönert, waren stets ein Ansporn für mich. Hilfreich zur Seite stand mir Angela Himmelreich, die Wörter fand, die es wirklich „nicht gibt“.

Danken möchte ich der MyCoRe-Community, zu deren Mitglied ich mich seit Januar 2003 zählen kann. Ich hoffe, ich lasse unsere gemeinsame Arbeit nicht all zu schlecht aussehen. Ich hoffe auch, dass die Arbeit kein Papiertiger ist, sondern sich für die Community als wichtiger Meilenstein erweist.

Bei meinen Freunden, die in den letzten sechs Monaten wenig Kontakt mit mir hatten, möchte ich mich bedanken, dass wir Freunde geblieben sind. Bei meiner Familie bedanke ich mich für die Unterstützung, die mir während meines Studiums zu Teil wurde. Meinen Verwandten, besonders meinen Großeltern, danke ich für die finanzielle Unterstützung während des Studium. Szilvia, meiner Frau, gebührt jedoch der größte Dank, ohne ihre Unterstützung hätte ich mein Studium niemals beenden können.

Diese Arbeit ist Gerhard Scheffler († 29.07.2000) gewidmet.

Inhaltsverzeichnis

Vorwort	vii
Inhaltsverzeichnis	xi
Tabellenverzeichnis	xiii
Abbildungsverzeichnis	xv
Codebeispiele	xvii
1 Einleitung	1
2 Grundlagen eines Dokumentenservers	3
2.1 Eine Einführung	3
2.2 DINI - Dokumentenserver	6
2.3 Begriffsklärung	8
3 Analyse MyCoRe-Persistenz	13
3.1 MyCoRe: Einführung	13
3.2 Das MCRObject	17
3.2.1 MyCoRe-Object: XML-Repräsentation	17
3.2.2 Von XML zum JAVA-Objekt	20
3.3 Das MCRDerivate	22
3.4 Die MCRClassification	27
3.5 Suche in MyCore	28
3.6 Persistenz	30
3.6.1 Persistenz von <mycoreobject>	30
3.6.2 Persistenz von <mycorederivate>	37
3.6.3 Persistenz von <mycoreclass>	40
3.6.4 Fazit	45

4	Content Repository Schnittstelle: JSR-170	49
4.1	Einführung	49
4.2	Repository-Modell	51
4.2.1	Level 1 Funktionen	55
4.2.2	Level 2 Funktionen	58
4.2.3	Optionale Funktionen	60
4.3	Knotentypen	61
4.3.1	Knotentypfunktionen in JCR-Levels	62
4.3.2	Definition von Knotentypen	62
4.4	XML Import und Export	68
4.4.1	Systemsicht	68
4.4.2	Dokumentsicht	70
4.5	XPATH-Suche in JCR	71
4.5.1	Aufbau der Suchanfrage	71
4.5.2	Einschränkungen gegenüber XPATH	72
4.5.3	Erweiterungen gegenüber XPATH	73
4.6	Fazit	75
5	Vergleich der Architekturmodelle	77
5.1	Vergleich der Persistenzfunktionen	77
5.2	Vergleich der Architekturen	80
6	Architekturentwurf für ein MyCoRe Next	83
6.1	Diskussion: MyCoRe auf JCR?	83
6.2	Abbildung des MyCoRe-Datenmodels auf JCR	85
6.2.1	Portierung mit geringstmöglicher Änderung	86
6.2.2	Portierung mit Nutzung der UUID	87
6.2.3	Portierung mit Nutzung von Hierarchien	91
6.2.4	Portierung der Metadatentypen von MyCoRe 1.0	94
6.3	Architekturkonzepte	95
6.4	Fazit	97
7	Zusammenfassung und Ausblick	99
A	Metadatentypen aus MyCoRe 1.0	101
B	Knotentypen aus JCR 1.0	105
B.1	Primäre Knotentypen	105
B.2	„Mixin“ Knotentypen	106
	Literaturverzeichnis	109

Tabellenverzeichnis

3.1	Aufbau der <code>MCRObjectID</code>	18
3.2	Bestandteile des <code>MCRObject</code>	20
3.3	Bestandteile des <code>MCRDerivate</code>	25
4.1	Vordefinierte Namenräume in JSR-170	57
4.2	Knotentypdefinition: <code>nt:base</code>	65
5.1	Funktionsmatrix: Persistenz von MyCoRe und JCR	79
6.1	Knotentypdefinition: <code>mcrnt:object</code>	89
6.2	Knotentypdefinition: <code>mcrnt:hierarchyObject</code>	92
6.3	Knotentypdefinition: <code>mcrmix:checkable</code>	94
A.1	Vordefinierte Metadatentypen in MyCoRe Version 1.0	104
B.1	Knotentypdefinition: <code>nt:unstructured</code>	106
B.2	Knotentypdefinition: <code>mix:referenceable</code>	107
B.3	Knotentypdefinition: <code>mix:versionable</code>	108

Abbildungsverzeichnis

3.1	Architektur des MyCoRe-Systems (Entwurf)	14
3.2	Klassendiagramm: MyCoRe-Typen	16
3.3	Klassendiagramm für <metadata>	22
3.4	Klassendiagramm für <derivate>	25
3.5	Klassenhierarchie MCRBase	26
3.6	createInDatastore-Methode von MCRObjekt	33
3.7	receiveFromDatastore-Methode von MCRObjekt	34
3.8	Klassendiagramm: MCRXMLTableManager und MCRXMLTableInterface	35
3.9	Klassendiagramm: MCRObjektSearchStoreInterface	35
3.10	deleteFromDatastore-Methode von MCRObjekt	36
3.11	Klassendiagramm: Internal File System	38
3.12	Klassendiagramm: IFS	39
3.13	Klassendiagramm: MCRClassification	41
3.14	Klassendiagramm: MCRClassificationObject	42
3.15	Klassendiagramm: MCRClassificationManager	43
3.16	Architektur des MyCoRe-Systems (Realisierung)	46
4.1	Klassendiagramm: Item, Node und Property	52
4.2	Hierarchie: Arbeitsbereich	53
4.3	Klassendiagramm: Session, Workspace, Elemente und Werte	54
4.4	Klassendiagramm: Zugriffskontrolle in JCR	58
4.5	Klassendiagramm: JCR-Knotentypen	67
6.1	Architektur des MyCoRe auf JCR	85
6.2	Logische Hierarchie von MyCoRe-Dokumenten	86
6.3	JCR-Sicht der Dokumenten-Hierarchie (Model 1)	87
6.4	JCR-Sicht der Dokumenten-Hierarchie (Model 2)	91
6.5	JCR-Sicht der Dokumenten-Hierarchie (Model 3)	93

Codebeispiele

3.1	XML-Darstellung eines MyCoRe-Objekts	17
3.2	Datenmodell-Definition des Typs: sample	19
3.3	XML-Darstellung eines MyCoRe-Derivats	24
3.4	XML-Darstellung einer MyCoRe-Klassifikation	27
3.5	XML-Darstellung eines Suchergebnisses	29
3.6	setFromJDOM()-Methode von MCRClassification	44
4.1	XML-Dokumentsicht des Knotentyps nt:base	66
4.2	Beispiel eines XML-Exports in Systemsicht	69
4.3	Beispiel eines XML-Exports in Dokumentsicht	70

Kapitel 1

Einleitung

Umberto Eco wurde zu einem Symposium der Universität von Jerusalem eingeladen. Thema war: „Das Bild von Jerusalem und dem Tempel als ein Bild der Geschichte.“ Eco selbst konnte mit diesem Thema recht wenig anfangen. Da er seine Dissertation über Thomas Aquinas (auch bekannt als: Thomas von Aquin), einem berühmten Theologen des Mittelalters, geschrieben hat und über dessen gesammelte Werke verfügte, schaute er im Index all dieser Bücher nach dem Term „Jerusalem“. Es war ein guter Ausgangspunkt, um sich mit dem Bild von Thomas Aquinas über Jerusalem vertraut zu machen. Mit nur diesen Büchern kam er auf 10 bis 15 Quellen für das Wort Jerusalem, die er hätte untersuchen müssen. *„Unfortunately I now have the Aquinas hypertext... and there I found, that there where [...] round 10,000 or so tokens...“* (Eco u. Coppock 1995)

Es spielt also eine Rolle, wie wir mit Wissen im so genannten „Informationszeitalter“ umgehen. Früher waren Bibliotheken Anlaufpunkt zu allen Wissensfragen. Heute bietet oft das Internet eine Alternative, um schnell zu „Wissen“ zu gelangen. Wir leben in einer Zeit, in der googeln ein Wort der deutschen Sprache geworden ist – der natürliche Weg der Wissensbeschaffung sozusagen. Mit dem exponentiellen Wachsen des Internets steigt auch die durchschnittliche Trefferzahl bei den Suchmaschinen. Längst sind diese bei den häufigsten Begriffen größer als die von Eco nicht mehr beherrschbaren 10.000 Treffer.

An diesem Punkt sind es die Bibliotheken, die sich ihrer alten Rolle als Wissensvermittler wieder bewußt geworden sind. Sie bieten verstärkt ganze Literaturarchive online für jedermann oder ausgewählte Benutzerkreise zur Recherche an. Dabei filtern Sie das Wissen durch Schlagwortvergabe und mit Hilfe von Stichworten. Den Bibliotheksnutzern geben sie Recherchehilfsmittel zur Hand, um die Informationsmengen weiter zu verarbeiten. Historische Aufgaben der Bibliothek haben ihr den Weg in das Informationszeitalter gebahnt. Im Zuge dieser Umstellungen sind viele Bibliothekssysteme entstanden, die diese

bei ihren Aufgaben unterstützen sollen. Diese Systeme aktuell zu halten, dient also nicht nur dem Interesse der Bibliothek sondern vor allem den Nutzern, die vorhandenes Wissen nutzen wollen, um es ausbauen zu können. Doch mit steigender Anzahl an zu bewältigenden Informationen stoßen diese Systeme an ihre Grenzen. Das schließt Leistungsschranken, aber auch funktionale Beschränkungen ein. MyCoRe, als Gegenstand dieser Arbeit, ist eines dieser Systeme und wird von zahlreichen Universitäten in Deutschland entwickelt und eingesetzt. Die Untersuchung der Leistungsgrenzen erfolgte bereits in (Bender 2005) und so widmet sich diese Arbeit den funktionellen Beschränkungen von MyCoRe.

Wie die Arbeit aufgebaut ist

Die Arbeit besteht aus sieben Kapiteln und zwei Anhängen.

Kapitel 2 (Grundlagen eines Dokumentenservers) deckt die Grundlagen dieser Arbeit ab und dient zur Einführung für die nachfolgenden Kapitel. **Kapitel 3 (Analyse MyCoRe-Persistenz)** liefert neben einer umfassenden Einführung in das MyCoRe-System die Analyse der Persistenzfunktionen von MyCoRe. Dieses Kapitel ist in sich abgeschlossen und kann daher getrennt von den anderen Kapitel gelesen werden. **Kapitel 4 (Content Repository Schnittstelle: JSR-170)** stellt als zweiten Standard die *Java Content Repository API* vor und ist ebenfalls in sich geschlossen. **Kapitel 5 (Vergleich der Architekturmodelle)** baut auf den vorigen zwei Kapiteln auf und stellt beide Systeme in einem kurzen Vergleich gegenüber. **Kapitel 6 (Architekturentwurf für ein MyCoRe Next)** zeigt ausgehend von den Erfahrungen aus Kapitel 3 und 4 Lösungsmöglichkeiten, um MyCoRe noch flexibler und leistungsfähiger zu machen. **Kapitel 7 (Zusammenfassung und Ausblick)** dient der Reflektierung der Arbeit und gibt einen Ausblick in mögliche zukünftige Entwicklungen.

Kapitel 2

Grundlagen eines Dokumentenservers

In diesem Kapitel sollen die Grundlagen eines Dokumentenservers geklärt werden. Doch es ist nicht sofort klar, was hinter dem Begriff „Dokumentenserver“ überhaupt steckt.

2.1 Eine Einführung

Ein Dokumentenserver oder Online-Archiv ist ein Internet-Dienst, auf dem Elektronische Publikationen veröffentlicht und archiviert werden können. Im Gegensatz zu einer einfachen Publikation auf einer Homepage, sorgt der Betreiber des Publikationsservers auch für die Langzeitarchivierung und Erschließung der publizierten Dokumente mit Hilfe von Metadaten. Häufig werden Dokumentenserver für Preprints verwendet.

(Wikimedia Foundation 2005)

Für eine Einleitung mag diese Definition reichen, sie gebraucht jedoch eine Reihe weiterer Begriffe, die es zunächst zu klären bedarf (akademie.de 2005).

Ein Dokument stellt man im Zusammenhang mit elektronischer Datenverarbeitung als Dateien dar, die nicht ausführbar sind. Dazu gehören einfache Textdateien, Bilder oder zum Beispiel HTML-Seiten. Es liegt in einer anderen semantischen Ebene, als der Dateibegriff, so dass ein Dokument durch mehrere Dateien repräsentiert werden kann oder auch mehrere Dokumente in einer Datei vorhanden sein können. Keinesfalls darf man Datei mit Dokument gleichsetzen, wenn auch eine Verwandtschaft zweifelsfrei besteht.

Zur Verwaltung elektronischer Dokumente werden oftmals Content-Management-Systeme eingesetzt.

Inhalt nagement-Systeme eingesetzt. Dabei handelt es sich um Software, die Informationsangebote verwaltet, zum Beispiel Inhalte von Webseiten. Inhalt wird als abstrakter Begriff verwendet und bezeichnet alles, was – auch im übertragenen Sinne – „etwas anderes füllt“. Im konkreten Fall kann der Inhalt eines Buches gemeint sein. Bei diesen abstrakten Inhalten handelt es sich meist um Informationen oder Wissen. Zwischen beiden Begriffen wird unterschieden. *Information* ist ein vielgebrauchter Begriff. Zum einen dient er der statistischen Untersuchung eines Informationsträgers, zum Beispiel bei der Kryptoanalyse. Aber auch Struktur/Syntax und Semantik bilden verschiedene Ebenen des Informationsbegriffs. Allgemein versteht man unter Informationen etwas Übertragbares¹, das beim Empfänger eine Zustandsänderung bewirkt. *Wissen* ist ein innerer Zustand (eines Menschen), der durch Informationen beeinflusst wird, in dem sein Muster in einem bestimmten Kontext interpretiert wird.

Als Abgrenzung zu Content-Management-Systemen existieren sowohl der Spezialfall Web-Content-Management-System (WCMS), als auch die Erweiterung Enterprise-Content-Management-System (ECMS). Web-Content-Management-Systeme sind Content-Management-Systeme, die überwiegend für die dynamische Publikation von Webseiten verwendet werden. In einigen Fällen werden CMS und WCMS synonym gebraucht, jedoch sind CMS im Gegensatz zu WCMS nicht auf das Internet als Einsatzort beschränkt. Ein WCMS kann Teil eines Enterprise-Content-Management-Systems sein.

„Enterprise Content Management sind die Technologien, Werkzeuge und Methoden zur Erfassung, Verwaltung, Speicherung, Bewahrung und Bereitstellung von elektronischen Inhalten im ganzen Unternehmen.“

(AIIM 2005, übersetzt aus dem Englischen)

Typischerweise umfasst ECMS eine ganze Reihe verschiedener Technologien, wie Dokumenten-Management, Kollaboration, Web-Content-Management, Workflow, Speicherung und Archivierung. Vorwiegend werden von ECMS schwach- oder unstrukturierte Informationen verwaltet, die man als Dokumente oder auch „neudeutsch“ Content bezeichnet. Dabei ist eine direkte Übersetzung von Content in Inhalt nicht korrekt. In seiner gebräuchlichen Nutzung umfasst der Begriff sowohl Inhalt wie auch Metadaten. Metadaten bezeichnen allgemein Daten, die Informationen zu anderen Daten enthalten. Während der Begriff der Metadaten erst in neuerer Zeit gebräuchlich ist, ist das Sammeln von „Metadaten“ seit Beginn des Bibliothekswesens gängige Praxis. So werden zu Büchern zum Beispiel Daten wie Autor und Erscheinungsjahr erfasst.

Allen Content-Management-Systemen ist die Trennung von Funktion, Content und Präsentation gemeinsam. Zu den Kernfunktionen eines CMS zählen die

¹zum Beispiel eine Nachricht

Erstellung² von Content, seine Verwaltung³ und Bereitstellung⁴, die Kontrolle⁵ von Content und seiner Individualisierung⁶.

Da nun die begrifflichen Grundlagen eines Dokumenten-Servers geklärt sind, ergibt sich nach der Definition von Seite 3, dass diese eine spezielle Form von Content-Management-Systemen darstellen. Weitere (komplexere) Vertreter von (Enterprise-)Content-Management-Systemen sind Online-Publishing-Systeme, die in (Krebs 2004) vorgestellt werden.

²auch indirekt, durch Anbindung externer Programme

³dem *Content-Management*

⁴Präsentation, Distribution

⁵Versionierung, Rechteverwaltung und Zugriffsregelungen

⁶Personalisierung von Content und Sichten darauf

2.2 DINI - Dokumentenserver

Während die Akzeptanz elektronischer Medien immer weiter zunimmt, existieren noch weitestgehend Defizite bei der Distribution von Forschungsveröffentlichungen. Plattform für diese Veröffentlichungen sind bislang die Verlagspublikationen gewesen, deren Preise mit dem Wachsen ihrer Monopolstellung stiegen. Ziel der DINI⁷-Initiative ist es, dieses Quasi-Monopol mit Hilfe von Dokumenten- und Publikationsservern zu durchbrechen. Der Einsatz von verteilten Dokumentenservern zwingt jedoch diese dazu, gewisse Mindeststandards einzuhalten, damit Daten recherchierbar sind. Besonders bislang kommerziell wenig interessante Publikationen können von dieser Initiative profitieren.

In (AG Elektronisches Publizieren 2003) werden Bedingungen definiert, die Dokumentenserver erfüllen müssen, um das sogenannte „DINI-Zertifikat“ zu bekommen. Das Zertifikat dient als Qualitätsmerkmal eines Dokumentenservers und soll sicherstellen, dass die Kommunikation über alle DINI-konformen Server möglich ist, um eine einheitliche Informationsstruktur zu schaffen. Im folgenden werden diese Bedingungen beschrieben.

Leitlinien

Als Betreiber eines Dokumentenservers wird man verpflichtet, Leitlinien (so genannte *Policies*) zu erstellen. In diesen werden inhaltliche Kriterien und Kriterien für den Betrieb eines Servers erfasst. Diese Leitlinien müssen öffentlich zugänglich sein. Sie regeln ferner die Rechte und Pflichten sowohl des Betreibers, als auch der Autoren und Herausgeber. Es wird festgelegt, welcher inhaltlichen, funktionalen und technischen Qualität Dokumente entsprechen, die auf dem Dokumentenserver veröffentlicht werden. Ausgehend von diesen Qualitätsmerkmalen, werden verbindliche Zusagen über die Archivierungszeiträume getroffen, die seitens des Betreibers sichergestellt werden. Festlegungen zur Operation des Dokumentenservers, wie auch bestimmter Leistungen, die gegenüber Autoren und Herausgebern angeboten werden, gehören ebenfalls in die Leitlinien. Neben weiteren organisatorischen und rechtlichen Aspekten, die in diesen Leitlinien aufgenommen werden müssen, existieren auch eine Reihe von technischen Anforderungen.

Authentizität und Integrität

Der Betreiber eines Dokumentenservers muss sowohl die Sicherheit des Servers als auch der Dokumente garantieren. Neben einer Dokumentation des technischen Systems, gehört auch die Sicherung der Dokumente und Metadaten dazu. Der Dokumentenserver sollte zudem so ausgelegt sein, dass er stets online erreichbar ist. Es muss außerdem sichergestellt und nachvollzogen werden können, dass Dokumente eingestellt wurden. Zum Zweck der Dokumentsicherheit bekommt jedes Dokument einen dauerhaften Bezeichner (*Persistent Identifier*),

⁷Deutsche Initiative für Netzwerkinformation: <http://www.dini.de/>

der eine Version eines Dokumentes eindeutig beschreibt. Ein geändertes Dokument bekommt also einen eigenen Bezeichner. Alle eingereichten Dateien eines Autors, werden unabhängig von eventuellen Konvertierungen, immer im Einreichungsformat abgespeichert. Vorkehrungen, die als Nachweis der Unversehrtheit eines Dokumentes dienen, sind zudem wünschenswert. Dies kann über Prüfsummen, oder digitale Signaturen (zum Beispiel nach § 2 SigG 2001) realisiert werden.

Das Erschließen von Dokumenten gehört seit jeher zu den Aufgaben eines Bibliothekars. Es sorgt dafür, dass diese in späteren Literaturrecherchen wiedergefunden werden können. Für das DINI-Zertifikat muss ein Leitfaden zur Sacherschließung vorhanden sein. Als Mindestanforderung müssen die Dokumente verbal durch freie Schlagwörter oder Klassifikationen erschlossen werden. Es wird empfohlen dafür wenigstens die Dewey-Dezimalklassifikation (DDC) als allgemeine und zusätzlich eine weitere gängige, allgemeine oder fachspezifische Klassifikation zu nehmen (Schlagwortnormdatei, PACS⁸ etc.). Auch die Erschließung mittels englischer Schlagwörter und Kurzfassung wird vorgeschlagen. *Sacherschließung*

Damit Daten an andere Systeme übermittelt werden können, müssen die Metadaten frei zugänglich und exportierbar sein. Mindestens der „Dublin Core Simple“ Standard (NISO 2001) muss für den Export der Metadaten unterstützt werden. Für den Export von Metadaten existieren eine Reihe von Standards. Je mehr ein Server unterstützen kann, desto mehr kann er auch mit Nicht-DINI-Servern interagieren. Bei dem Metadaten-Export spricht man allein über das Format der Metadaten, den Zugriff darauf regeln die Schnittstellen. *Metadaten-Export*

Schnittstellen müssen nicht nur für den Nutzer bereit stehen, sondern auch für externe Systeme. Nutzer müssen über eine Web-Schnittstelle Zugang erhalten. Die Forderung nach dem Metadaten-Export zumindest nach dem DC-Simple-Standard wurde bereits erwähnt. Der Zugriff auf diese exportierten Daten muss auch über eine festgelegte Schnittstelle erfolgen. Hier muss mindestens OAI-PMH 2.0 (Lagoze u. a. 2002) unterstützt werden. OAI steht für *Open Archives Initiative* und ist ein international verwendeter Standard für den Export beliebiger Metadatenformate, verlangt aber mindestens den bereits besprochenen DC-Simple-Standard. Die Unterstützung von Web-Services mittels SOAP oder des verbreiteten Standards Z 39.50 (NISO 2003) ist optional. *Schnittstellen*

Jeder Dokumentenserver muss Zugriffsstatistiken im Rahmen geltender Gesetze (Datenschutz etc.) erstellen. Dokumentenbezogene Statistiken stellen ein Indiz für die inhaltliche Relevanz bezüglich anderer Dokumente dar, während serverbezogene Statistiken helfen können, die technologische Basis des Servers zu bewerten. An ihnen sind zum Beispiel etwaige Wartezeiten des Nutzers *Zugriffsstatistiken*

⁸Physics and Astronomy Classification Scheme

für eine Abfrage später sichtbar. Dies kann der Vergleichbarkeit verschiedener DINI-Server dienen.

Die Garantie der Langzeitverfügbarkeit von Metadaten und Dokumenten, zum Beispiel bei Zugriff über den dauerhaften Bezeichner, muss mittels einer Mindestzeit in den Leitlinien formuliert sein. Diese Mindestzeit darf fünf Jahre nicht unterschreiten. Die Aufgabe der Langzeitarchivierung kann an spezielle Archivierungsinstitutionen, zum Beispiel „Die Deutsche Bibliothek“⁹, ausgelagert werden und muss damit nicht direkt durch den Dokumentenserver sichergestellt werden.

Das DINI-Zertifikat regelt keine Implementierungen und stellt keine hohen Anforderungen. Ziel ist, dass eine möglichst große Zahl von Dokumentenservern zusammen einen großen Stand an Dokumenten bereitstellen, die von verschiedenen zentralen Stellen erreicht werden können, also von dort suchbar sind. Das Zertifikat ist ein Gütesiegel, das in Deutschland begehrt und gern gesehen ist. Ein Dokumentenserver sollte also eine Reihe von Funktionen mitbringen, um die Anforderung zu unterstützen. Zusammengefasst zählen dazu sowohl formale Dinge, wie Leitlinien und Autorenunterstützung als auch Software-Anforderungen, wie Exportmöglichkeiten in verschiedene Metadatenformate und eine breite Unterstützung externer Schnittstellen. MyCoRe, als ein möglicher Anwendungskern eines (DINI-)Dokumentenservers, wird in Kapitel 3 vorgestellt.

2.3 Begriffsklärung

In dieser Arbeit werden zwei Varianten eines Content-Management-Backends untersucht: MyCoRe und JCR (vgl. Kapitel 3 und 4). In diesem Umfeld haben sich bestimmte Begrifflichkeiten entwickelt, die teilweise nicht ganz denen des Abschnitts 2.1 entsprechen. Zur Klärung werden hier diese und andere Begriffe der vorliegenden Diplomarbeit definiert, während die Vorstellung in Abschnitt 2.1 eher als Einleitung verstanden werden soll, um die Arbeit einordnen zu können.

Objekte sind Informationseinheiten unterschiedlicher Ausprägung. Sie fassen eine Reihe von Informationen zusammen und repräsentieren meistens ein „Ding“ der realen Welt. Typische Objekte in Bibliothekssystemen sind Personen und Dokumente. Am Beispiel der Personen und Dokumente kann man deutlich machen, dass Objekte in Beziehung zu anderen Objekten stehen können, um konkrete Sachverhalte zu modellieren. In dem Beispiel könnte eine Person Autor eines Dokuments sein. Von Objekten

⁹<http://www.ddb.de/>

in diesem Sinne wird ausschließlich im Kontext von MyCoRe in dieser Arbeit gesprochen.

Datenmodell ist die schematische Definition des Aufbaus von Objekten, so dass diese automatisch verarbeitet und geprüft werden können. Das Datenmodell sorgt dafür, dass essentielle Informationen immer vorhanden und gleich aufgebaut sind. Häufig wird in diesem Zusammenhang auch vom Metadatenmodell gesprochen, weil es sich bei den hier betrachteten Objekten in der Regel um Träger von Metadaten handelt.

Dokument bezeichnet eine spezielle Klasse von Objekten. Dokumente besitzen neben den angesprochenen Metadaten, für die ein Datenmodell definiert wurde, noch Dateien. Diese Dateien stellen Dokumente im herkömmlichen Sinne dar. Als Beispiel für solche Daten können HTML-Dokumente, PDF-Dokumente, oder JPEG-Bilder dienen.

Klassifikation bezeichnet die festgelegte Einteilung von Objekten in Klassen. Eine Klassifikation könnte zum Beispiel `FORMAT` heißen und ihre Klassen `FILM`, `TEXT`, `TON` und `BILD` etc.

Persistenz stellt sicher, dass Objekte nicht nur im laufenden System existieren, sondern unabhängig vom Status des Systems extern gespeichert werden. Die Synchronisation zwischen System und Persistenzschicht übernehmen spezielle Vermittler, die dafür sorgen, dass lokale Kopien im System den Objekten in der Persistenzschicht entsprechen. Diese Vermittler bieten neben reinen Lese- und Schreibzugriffen noch zusätzliche Funktionen an, die man unter dem Begriff „Persistenzfunktionen“ zusammenfasst. So unterscheidet man ferner flüchtige von persistenten Änderungen an Objekten. Bei Computern werden in der Regel flüchtige Änderungen im RAM, während persistente auf Externspeichern wie Festplatten vorgenommen werden. Bei einem Neustart des Systems/Computers sind flüchtige Änderungen verloren, im Gegensatz zu persistenten Änderungen.

Datenintegrität stellt zumindest sicher, dass Daten in der Persistenzschicht nicht verfälscht werden. Das heißt, nach einem Schreiben von Objekt A und einem anschließenden Lesen von Objekt A treten keine Unterschiede auf. Es gehen in der Persistenzschicht keine Informationen verloren. Neben der Datensicherheit können noch andere Anforderungen an die Persistenzschicht existieren. Ein Beispiel ist die referentielle Integrität, bei der sichergestellt wird, dass keine Zeiger auf andere Objekte ins „Leere“ zeigen können. Ein Spezialfall der referentiellen Integrität ist das „Waisenkinderverbot“, bei der ausschließlich sichergestellt wird, dass Kinderobjekte nicht ohne ihre Väter existieren. Ein Kapitelobjekt ist beispielsweise

ein Kind eines Buchobjekts. Bei Einhalten des Waisenkinderverbots kann dieses Kapitelobjekt nur so lange existieren, wie auch sein Vater, das Buchobjekt, existiert.

Content Repository ist eine Spezialform einer Persistenzschicht. In der Regel ist ein Content Repository anwendungsunabhängig. Es ist generisch entwickelt, um Anfragen über verschiedene Datentypen/Datenmodelle in gleicher Weise behandeln zu können. Zu solchen Anfragen könnte zum Beispiel gehören: „Zeige alle Objekte, die in der letzten Woche verändert worden sind!“ Dabei spielt es keine Rolle, ob diese Objekte zum Beispiel Autoren oder Dokumente sind und wie diese letztendlich physisch abgespeichert werden. Mit einem Content Repository arbeitet man also über eine abstrahierende Schnittstelle zusammen.

Content Management fasst Funktionen eines Content Repositories mit Funktionen für die Eingabe, die Aufbereitung und Präsentation der Objekte zusammen.

Versionierung stellt sicher, dass trotz Änderungen an Objekten später jeder ältere Zustand wieder herstellbar ist, um ausgehend von diesem wieder Änderungen vorzunehmen. Dabei kann es in der Regel auf drei verschiedene Arten zu neuen Versionen kommen (Lahnakoski 2005). *Historische Versionen* sind Versionen, die jeweils maximal eine Vorgängerversion und eine Nachfolgerversion besitzen. *Logische Versionen* entstehen, wenn ausgehend von einer Version eines Objektes verschiedene Versionen entstehen, beispielsweise wenn gleichzeitig an einem Objekt unterschiedliche Änderungen vorgenommen wurden (*split*). Dagegen kommt es zur einer *Zusammengesetzten Version*, wenn ausgehend von verschiedenen Versionen eine neue Version erstellt wird. Dies tritt zum Beispiel auf, wenn die unterschiedlichen Änderungen bei der logischen Versionierung in einer neuen Version zusammengeführt (*merge*) werden. Die Versions-History ist ein azyklischer Graph, bei dem die Knoten die Versionen und die (gerichteten) Kanten die Übergänge (Veränderungen) von einer Version zur anderen darstellen.

Zugriffsschutz (*access control*) stellt Möglichkeiten bereit, Nutzern bestimmte Aktionen auf Objekte zu erlauben oder zu verbieten. So dürfen bestimmte urheberrechtlich geschützte Werke ausschließlich nur bestimmten Personengruppen zugänglich gemacht werden, die jedoch keine Änderungen vornehmen dürfen.

Transaktion ist eine Folge von Aktionen über Objekten, die entweder komplett oder gar nicht ausgeführt werden (Atomarität). Beispielsweise gibt

es zwei Kontenobjekte, die Daten zum aktuellen Kontostand beinhalten. Von Konto A soll auf Konto B ein Geldbetrag transferiert werden. Dieser Vorgang als Transaktion stellt sicher, dass durch einen Fehler in der Änderung von Konto A oder B, weder Geld verloren geht (Fehler bei B), noch Geld „entsteht“ (Fehler bei A). Diese und weitere Eigenschaften einer Transaktion werden in (Häder u. Reuter 1983) erläutert.

Sitzung (*session*) ist die begrenzte Zeit, in der zwei Partner miteinander kommunizieren. Im Kontext einer Sitzung sind Zustände verwaltbar, während die Kommunikation zustandslos stattfindet. Es werden ausschließlich Nachrichten ausgetauscht, die beim Kommunikationspartner Aktionen auslösen.

Ereigniskontrolle (*event handling*) ermöglicht es, auf bestimmte Aktionen über Objekten zu reagieren. Zum Beispiel könnte beim Speichern von Objekten eine Email an eine bestimmte Person geschrieben werden. Ein Ereignis ist also das Auftreten einer Aktion in Bezug zu einem Objekt und Ereigniskontrolle das Mittel, um auf Ereignisse (Zustandsänderungen etc.) zu reagieren.

Kapitel 3

Analyse MyCoRe-Persistenz

Dieses Kapitel dient dazu, die Persistenzschicht von MyCoRe vorzustellen und insbesondere zu analysieren. Bevor die Möglichkeiten der Persistenz von Objekten analysiert werden kann, gilt es diese Objekte anhand von spezifischen Merkmalen zu unterscheiden. Die Abschnitte 3.2 – 3.4 stellen dabei die unterschiedlichen Klassen von Objekten vor. Es wird anhand von Beispielen erläutert, wie XML-Daten in JAVA-Objekte übersetzt werden und umgekehrt. In Abschnitt 3.6 wird dann jeweils erklärt, wie diese Daten dauerhaft gespeichert werden. Doch zunächst erfolgt in Abschnitt 3.1 eine kurze Einführung in das MyCoRe-System.

3.1 MyCoRe: Einführung

MyCoRe steht für „*My Content Repository*“. Es ist ein Produktkern für eigene Anwendungen und wurde für digitale Bibliothekssysteme und Archive entworfen. Ein Vorteil von MyCoRe ist das frei definierbare Datenmodell, so dass MyCoRe nicht prinzipiell auf den Einsatz in Bibliotheken beschränkt ist. Vielmehr stellt es Schnittstellen bereit, um Anwendungen, vorzugsweise Web-Applikationen, über ein selbst definiertes Metadatenmodell zu erstellen. Dabei können unterschiedliche Backends zum Einsatz kommen. Zu ihnen zählen kommerzielle Produkte, wie DB2¹ und der Content Manager V8.2² von IBM, oder Tamino³ der Software AG. Aber auch ein breites Angebot aus dem OpenSource-Bereich wird abgedeckt, z.B. mit MySQL⁴, Lucene⁵ und dem XML-

¹<http://www-306.ibm.com/software/data/db2/>

²<http://www-306.ibm.com/software/data/cm/cmgr/>

³<http://www.softwareag.com/tamino/>

⁴<http://www.mysql.com/>

⁵<http://lucene.apache.org/>

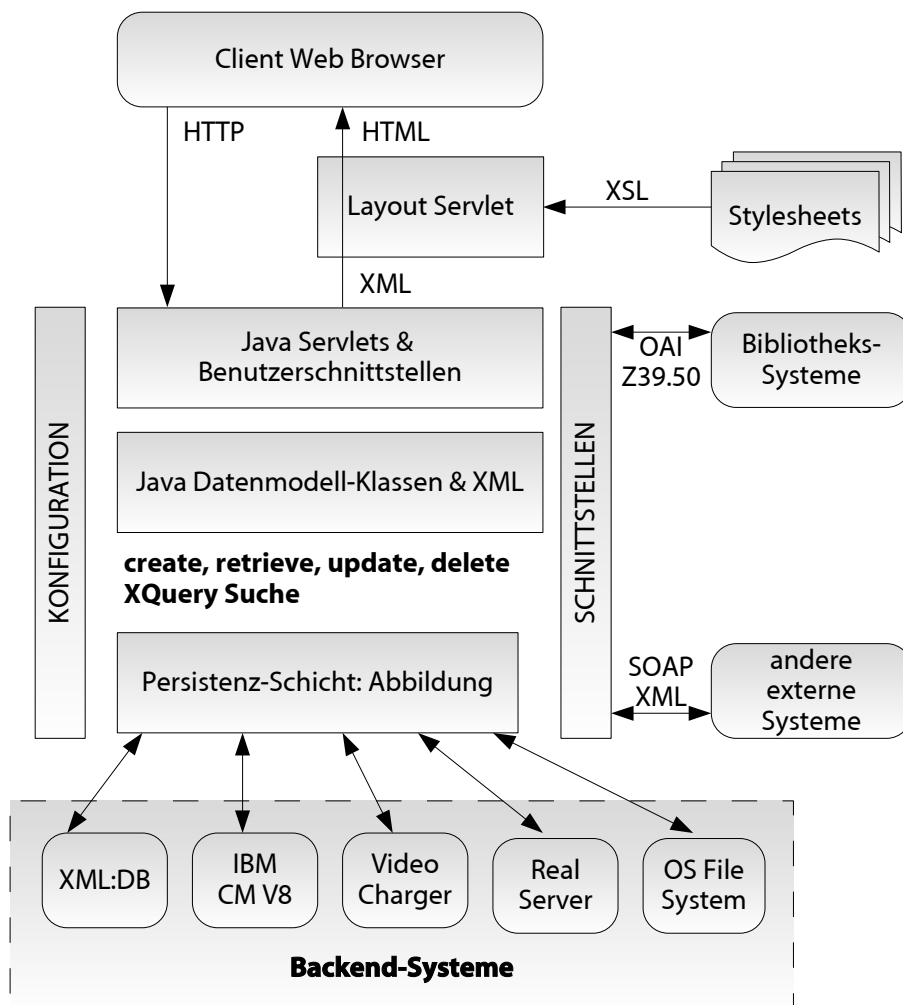


Abbildung 3.1: Architektur des MyCoRe-Systems (Entwurf)

Herstellerunabhängigkeit

DINI

Datenbanksystem eXist⁶. MyCoRe verdeckt gegenüber der Anwendung die zugrundeliegenden Softwarekomponenten, so dass eine schnelle Anwendungsentwicklung gefördert wird. Ein Wechsel der Softwarekomponenten beschränkt sich so im Idealfall auf das Exportieren der Daten, das Ändern von Konfigurationseinstellungen und den Reimport auf Basis der neuen Backends. Mittelfristig ist es ein Ziel der MyCoRe-Entwicklung einen Anwendungskern für Dokumentenserver zu liefern, der die Anforderungen des DINI-Zertifikats (vgl. Abschnitt 2.2) erfüllt. Die Auslegung MyCoRes Daten in XML zu verwalten begünstigt dabei den Metadatenexport, außerdem existieren eine Reihe von

⁶<http://exist.sourceforge.net/>

externen Schnittstellen für MyCoRe. Im Kern verfügbar ist zum Beispiel eine OAI-Implementierung (Lagoze u. a. 2002). Als *Persistent Identifier* unterstützt MyCoRe die Generierung von NBNs (*National Bibliography Number*).

Abbildung 3.1 zeigt die MyCoRe-Architektur nach (Lützenkirchen 2002) vorgestellte. Zu sehen ist, dass MyCoRe als Basis für eine Web-Anwendung entwickelt wurde, die die Nutzeroberfläche in einem Web-Browser darstellt. Der Browser kommuniziert mit der MyCoRe-Anwendung über HTTP-Requests, die von JAVA-Servlets verarbeitet werden. Herausgehoben aus dieser Gliederung ist das `MCRLayOutServlet`, das für die Transformation von XML-Daten, die MyCoRe zur Verarbeitung nutzt, in HTML-Seiten verantwortlich ist. Diese Transformation erfolgt mit Hilfe von XSL-Dateien (sog. Stylesheets). Neben der Web-Oberfläche existiert zudem ein Kommandozeilen-Interface, das Befehle im MyCoRe-System verarbeitet.

Die *Java Datenmodell-Klassen & XML-Schicht* beinhaltet Klassen, mit denen das flexible MyCoRe-Metadatensystem repräsentiert werden kann. In den Abschnitten 3.2, 3.3 und 3.4 werden die wichtigsten Vertreter vorgestellt.

Der Sinn von MyCoRe ist es, eine herstellerunabhängige Schicht über verschiedene Persistenzsysteme (hier: Backend-Systeme) zu bilden. So bekommt der Nutzer nichts davon mit, wo die Daten abgelegt sind, auf die er zugreifen kann. Der Administrator kann über Regeln definieren, welche Daten wo (und wie) abgespeichert werden. Insbesondere ist dabei die parallele Nutzung verschiedener Backend-Systeme eingeschlossen, wie sie in 3.6.2 näher beschrieben wird. Der Administrator entscheidet zum Beispiel auch für die Metadaten den Speicherort. Diese liegen intern wie erwähnt im XML-Format vor. Die *Persistenz-Schicht* kümmert sich darum, diese Darstellungsform in eine herstellerspezifische zu überführen. Für Metadaten werden eine XML:DB-kompatible Datenbank wie eXist oder Tamino verwendet oder der Content Manager V8.2 von IBM.

Die Kommunikation zwischen Datenmodellschicht und Persistenzschicht findet über eine Reihe von definierten Operationen statt. Ihre Implementierung wird in Abschnitt 3.6 näher untersucht. Rechts angeordnet in *Abbildung 3.1* sind Schnittstellen für externe Systeme aufgeführt, dies können andere Bibliothekssysteme sein oder gänzlich anwendungsfremde Systeme. Über eine mögliche Kooperation mittels Webservices beschäftigt sich eine andere Diplomarbeit (Krebs 2004) bereits.

Das Bild veranschaulicht, dass man grundsätzlich mit Konfigurationsdaten (u.a. Metamodell etc.) und XSL-Stylesheets zu seiner eigenen Anwendung kommen kann. Fehlende Funktionen können aber zusätzlich in JAVA implementiert und wieder in den MyCoRe-Kern zurückfließen. Der Kern wird fortwährend unter der **GNU General Public License** weiter entwickelt. Für die Betrachtung *eigene Anwendungen*

in dieser Diplomarbeit wurde die aktuelle MyCoRe Version 1.0 herangezogen. Ein Ziel dieser Arbeit soll sein, die bisherige Entwicklung kritisch zu hinterfragen und neue Perspektiven aufzuzeigen. Es gilt, wie einleitend erwähnt, das Vorurteil zu beseitigen – oder auch zu bestätigen – dass die Persistenzschicht wie ein Filter fungiert, der herstellspezifische Funktionen nicht nur maskiert (wie gewünscht), sondern wegfiltert und damit nicht nutzbar macht. Bevor allerdings die Persistenzschicht untersucht werden kann, bedarf es der Einführung, was eigentlich gespeichert werden soll.

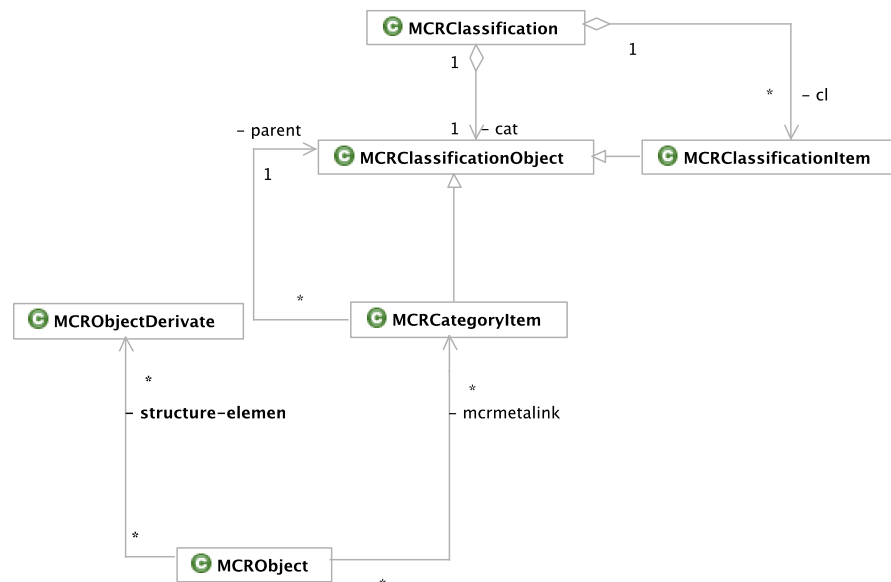


Abbildung 3.2: Klassendiagramm: MyCoRe-Typen

In **Abbildung 3.2** wird ein Ausschnitt der Datenmodell-Schicht von MyCoRe gezeigt. Es stellt die wichtigsten Klassen, die im folgenden beschrieben werden, dar. Das Diagramm lässt sich grob in drei Bereiche teilen. **MCRObjectDerivate** steht für den Datei-Teil in MyCoRe, also für die Dateien, die MyCoRe speichern und verwalten soll. **MCRObject** steht für die Metadaten (z.B. Titel, Autor etc.), die die Dateien näher beschreiben. Die übrigen Klassen sind für die Klassifikationsverwaltung von Belang. Jeder dieser drei Typen in MyCoRe, **MCRObject**, **MCRDerivate** und **MCRClassification**, wird in den folgenden drei Kapiteln vorgestellt. Anschließend erfolgt in Abschnitt 3.6 dann die Analyse der MyCoRe-Persistenzfunktionen auf Basis dieser Typen.

3.2 Das MCRObject

Für die nachfolgende Analyse der MyCoRe-Persistenzschicht ist es erforderlich, einen kurzen Exkurs in die Umsetzung einer XML-Präsentation in eine Reihe von JAVA-Objekten voranzustellen. Am Beispiel eines im praktischen Umfeld wohl kaum vorkommenden, aber leicht verständlichen Objektes soll diese Transformation erläutert werden.

3.2.1 MyCoRe-Object: XML-Repräsentation

Codebeispiel 3.1 XML-Darstellung eines MyCoRe-Objekts

```
<?xml version="1.0" encoding="utf-8"?>
<mycoreobject
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="datamodel-sample.xsd"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  ID="DiplArb_sample_1"
  label="Beispiel Objekt 1">
  <structure />
  <metadata>
    <titles class="MCRMetalangText" textsearch="true">
      <title xml:lang="de">
        Nur ein Beispielobjekt
      </title>
      <title xml:lang="en">
        Just a sample object
      </title>
    </titles>
    <creatorlinks class="MCRMetalinkID">
      <creatorlink xlink:type="locator"
        xlink:href="DiplArb_person_00000001"
        xlink:label="Author"
        xlink:title="Thomas Scheffler"/>
    </creatorlinks>
  </metadata>
</service/>
</mycoreobject>
```

Das Objekt in **Codebeispiel 3.1** hat die ID `DiplArb_sample_1`. Die MCRObjectID oder kurz ObjectID ist ein zusammengesetzter eindeutiger Be- *Objektbezeichner* zeichner für das gesamte Objekt. **Tabelle 3.1** zeigt den Aufbau einer solchen ObjectID.

Die Projekt-ID ist ein Bezeichner für die Anwendung, in der das MyCoRe-Objekt genutzt wird, z.B. DocPortal⁷. Objekt-Typen kann es ganz unterschiedliche geben, in **Codebeispiel 3.1** werden zwei benutzt: `sample` und `person`.

⁷<http://www.uni-leipzig.de/urz/leilib/docportal.htm>

Bestandteil	Beispiel
Projekt-ID	DiplArb
Objekt-Typ	sample
Zahl	1

Tabelle 3.1: Aufbau der MCRObjectID

Es ist dabei hervorzuheben, dass diese Objekttypen nur im Zusammenhang mit der Projekt-ID eindeutig sind, ein `person` für Projekt `DiplArb` kann somit andere – inkompatible – Metadaten besitzen, als `person` eines anderen Projekts. Im *User Guide* (Lützenkirchen u. a. 2005) ist erklärt, wie verschiedene Typen spezifiziert werden. Aus diesen Spezifikationen (z.B. **Codebeispiel 3.2**) werden durch MyCoRe-eigene Werkzeuge XML-Schemata erstellt (im Beispiel **datamodel-sample.xsd**), gegen die die XML-Dateien dann validiert werden können.

Es gibt aber einen Grundaufbau, der bei allen MyCoRe-Objekten gleich ist. Dazu gehört:

1. Das Wurzelement `<mycoreobject>` mit den Angaben für das verwendete Schema, der ID und einem Label.
2. Das `<structure>`-Element, das Informationen über mögliche Kinder oder einen Vater enthält.
3. Das `<metadata>`-Element, in dem die selbstdefinierten Metadaten gespeichert werden. Unterhalb dieses Elements ist nur der strukturelle Aufbau bestimmt. Die nächste Tag-Ebene (`<titles>` und `<creatorlinks>`) bestimmt von welchen Typ die Metadatenknoten in der darunter befindlichen Hierarchieebene sind. In unserem Beispiel heißt dies, dass `<title>`-Elemente vom Typ `MCRMetaLangText` sind und `<creatorlink>`-Elemente vom Typ `MCRMetaLinkID`.
4. Das `<service>`-Element enthält variable Informationen, wie Zugriffszeiten, oder einfache ACL⁸-Regeln.

⁸Access Control List

Codebeispiel 3.2 Datenmodell-Definition des Typs: sample

```

<?xml version="1.0" encoding="iso-8859-1"?>
<configuration type="sample">
  <structure>
    <!-- Object Children -->
    <element name="children" minOccurs="0" maxOccurs="1"
      parasearch="false" textsearch="false">
      <mcrmetalinkid name="child" class="MCRMetaLinkID"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
    <!-- Object Parent -->
    <element name="parents" minOccurs="0" maxOccurs="1"
      parasearch="false" textsearch="false">
      <mcrmetalinkid name="parent" class="MCRMetaLinkID"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
    <!-- Derivate Objects -->
    <element name="derobjects" minOccurs="0" maxOccurs="1"
      parasearch="true" textsearch="false">
      <mcrmetalinkid name="derobject" class="MCRMetaLinkID"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
  </structure>
  <metadata>
    <!-- 01 - Titel -->
    <element name="titles" minOccurs="1" maxOccurs="1"
      parasearch="true" textsearch="true">
      <mcrmetalangtext name="title" class="MCRMetaLangText"
        minOccurs="1" maxOccurs="unbounded" length="1024" />
    </element>
    <!-- 02 - CreatorLink -->
    <element name="creatorlinks" minOccurs="0" maxOccurs="1"
      parasearch="true" textsearch="false">
      <mcrmetalinkid name="creatorlink" class="MCRMetaLinkID"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
    <!-- 04 - Zielgruppe -->
    <element name="audiences" minOccurs="0" maxOccurs="1"
      parasearch="true" textsearch="false">
      <mcrmetaclassification name="audience"
        class="MCRMetaClassification"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
  </metadata>
  <service>
    <!-- Service Dates -->
    <element name="servdates" minOccurs="0" maxOccurs="1"
      parasearch="true" textsearch="false">
      <mcrmetadate name="servdate" class="MCRMetaDate"
        minOccurs="1" maxOccurs="unbounded" />
    </element>
    <!-- Service Flags -->
    <element name="servflags" minOccurs="0" maxOccurs="1"
      parasearch="true" textsearch="false">
      <mcrmetalangtext name="servflag"
        class="MCRMetaLangText"
        minOccurs="1" maxOccurs="unbounded" length="256" />
    </element>
  </service>
</configuration>

```

3.2.2 Von XML zum JAVA-Objekt

Auch wenn die Metadaten grundsätzlich in XML vorliegen und abgespeichert werden, verlangt JAVA eine objektorientierte Repräsentation dieser Daten. In diesem Abschnitt sollen deshalb Klassen vorgestellt werden, die diese Aufgabe für MyCoRe erfüllen. Dabei kommt es nicht zu sehr auf den Klassenaufbau an, welche Methoden aufgerufen werden, mit was für Parametern und welchen Rückgabewerten. Dies wird näher in Abschnitt 3.6 beschrieben. Vielmehr dient dieser Abschnitt als Überblick über beteiligte Klassen und verdeutlicht die Abbildung von XML zu JAVA-Objekten.

Klasse: `MCRObject`

Orientiert an der Struktur des XML-Dokuments ist auch das JAVA-Objekt `MCRObject` aufgebaut. Es ist abgeleitet von der abstrakten Klasse `MCRBase`, die wie alle in diesem Abschnitt besprochenen Klassen im Package `org.mycore.datamodel.metadata` liegen. Dabei stellt `MCRBase` die Basis für `MCRObject` und `MCRDerivate` bereit (siehe Abb. 3.5) und initialisiert den `MCRXMLTableManager`. Dieser ist ein Teil der Persistenzschnittstellen in MyCoRe und stellt Zugriffe auf XML-Ebene bereit. Beim Erstellen eines `MCRObject` kommt er aber nur dann zum Zuge, wenn der XML-Code über das `MCRXMLTableInterface` aus einem Backend geholt wird. Zur Zeit existiert nur eine Implementierung dieser Schnittstelle, im Package `org.mycore.backends.sql` die Klasse `MCRSQLXMLStore`.

Der erste Schritt, ein `MCRObject` zu erzeugen, führt über den XML-Code, der spätestens⁹ in der Klasse in ein JDOM-Dokument überführt wird, das, wie in `MCRBase` definiert, als „`jdom_document`“ benannt wird. Für die Schaffung der Struktur wird danach die Methode `set()` benötigt, die aus dem JDOM-Dokument weitere Objekte erzeugt und das `MCRObject` vervollständigt.

Datenfeld	Typ	Basisklasse
<code>mcr_id</code>	<code>MCRObjectID</code>	<code>MCRBase</code>
<code>mcr_label</code>	<code>java.lang.String</code>	<code>MCRBase</code>
<code>mcr_schema</code>	<code>java.lang.String</code>	<code>MCRBase</code>
<code>mcr_struct</code>	<code>MCRObjectStructure</code>	<code>MCRObject</code>
<code>mcr_metadata</code>	<code>MCRObjectMetadata</code>	<code>MCRObject</code>
<code>mcr_service</code>	<code>MCRObjectService</code>	<code>MCRBase</code>

Tabelle 3.2: Bestandteile des `MCRObject`

Tabelle 3.2 zeigt eine Übersicht dieser Objekte. Anhand dieser ist der enge Zusammenhang zwischen XML-Code und Klassenstruktur ersichtlich.

⁹falls nicht die Methode `setFromJDOM()` benutzt wird

Während `mcr_label` und `mcr_schema` nicht weiter syntaktisch untersucht werden, gibt es für die anderen Elemente jeweils eigene Klassen, die den korrekten Aufbau (siehe auch Tabelle 3.1) sicherstellen.

Das Interessanteste stellt dabei das Objekt `mcr_metadata` dar. Es ist noch vor `mcr_service` dasjenige, das die meisten Erweiterungsmöglichkeiten bietet, weil es beliebige Datentypen enthalten kann. Der Anwendungsentwickler kann entscheiden, ob die von MyCoRe bereitgestellten Typen ausreichend sind oder selbst entworfene genutzt werden sollen. *Freiheiten bei Metadaten*

Klassen: `MCRObjectMetadata` und `MCRMetaElement`

Es wird in diesem Abschnitt deutlich, wie die Trennung von XML-Tag-Hierarchien einerseits und die Arbeitsteilung der Klassen in `org.mycore.datamodel.metadata` andererseits erfolgt. Während `MCRObject` exklusiv auf das Wurzelement `<mycoreobject>` zugreift, werden die Tags der 1. Hierarchieebene (`<structure>`, `<metadata>`, `<service>`) von speziellen Klassen verwaltet.

Das Beispiel `MCRObjectMetadata` zeigt, dass diese Arbeitsteilung nicht in der 1. Hierarchieebene aufhören muss. So greift `MCRObjectMetadata` nur auf das `<metadata>`-Element selbst zu und überlässt der Klasse `MCRMetaElement` den Zugriff auf dessen Subelemente. Abbildung 3.3 zeigt ein Klassendiagramm der in diesem Abschnitt behandelten Klassen.

In Codebeispiel 3.1 sind unterhalb von `<metadata>` zwei weitere Tags (`<titles>`, `<creatorlinks>`). Sie enthalten das Attribut `class`. Es gibt an, welche JAVA-Klasse für die Validierung der enthaltenen Tags verantwortlich ist. So werden `<title>`-Tags von der Klasse `MCRMetaLangText` geparkt.

Dieser Mechanismus gestattet es, das Metadatenmodell um beliebige und beliebig viele Datentypen zu erweitern. Anhang A gibt einen Einblick über die in MyCoRe 1.0 enthaltenen Metadatentypen.

Die Abarbeitungsfolge ist dabei bei jeder `class`-Angabe gleich:

1. `MCRObjectMetaData` erstellt für jedes Untertag von `<metadata>` (in Codebeispiel 3.1 `<titles>` und `<creatorlinks>`) eine Instanz von `MCRMetaElement` und übergibt der Methode `setFromDOM(org.jdom.Element)` dieses zur weiteren Verarbeitung.
2. `MCRMetaElement` verwaltet, ob Metadaten vererbt werden können oder geerbt werden und ob sie für Volltextsuche (zum Beispiel bei Beschreibungen) oder parametrische Suche (zum Beispiel nach einem Datum) freigeschaltet werden.

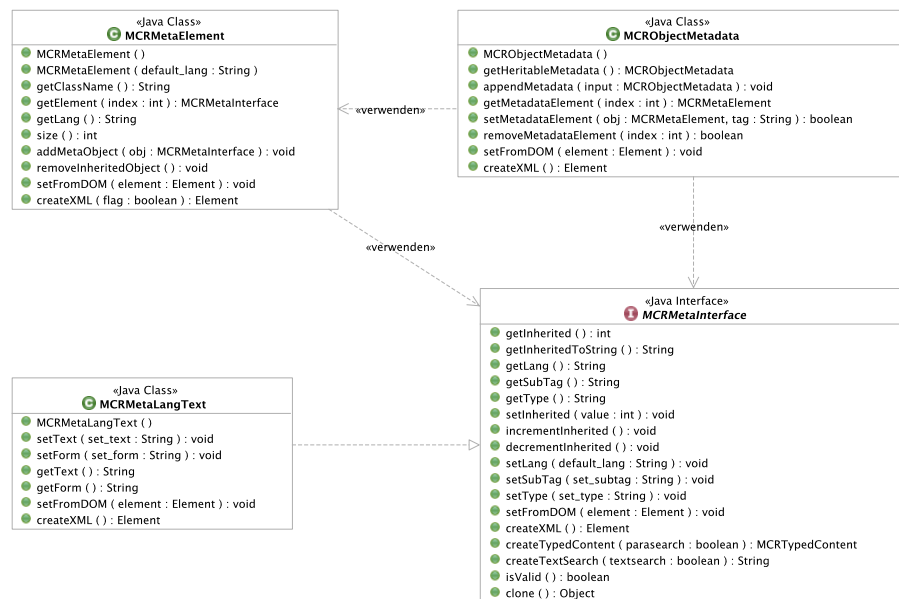


Abbildung 3.3: Klassenhierarchie für das <metadata>-Element

Ferner liest sie das `class`-Attribut aus und erstellt für jedes Untertag (zum Beispiel <title> bei <titles>) eine neue Instanz der angegebenen Klasse und übergibt dieses Element der Methode `setFromDom(org.jdom.Element)`.

3. `MCRMetaLangText` (laut Beispiel) implementiert die Schnittstelle `MCRMetaInterface` und erweitert die Klasse `MCRMetaDefault`, in der grundlegende generische Arbeitsschritte ausgelagert sind.

Sowohl `MCRObjectMetaData` und `MCRMetaElement` führen intern eine Liste aller erstellten Objekte und diese geben über die Methode `createXML()` dann ein `org.jdom.Element` zurück, die für `MCRObjectMetaData` das komplette <metadata>-Element wiederspiegelt.

3.3 Das MCRDerivate

Im Allgemeinen versteht man unter einem Derivat (lat.) eine Sache, die von einer anderen Sache abgeleitet ist. In der Chemie sind das Stoffe (Atome und Moleküle), die von anderen abgeleitet sind und eine ähnliche Struktur besitzen. Auch in anderen Bereichen der Wissenschaft und der Industrie ist das Wort Derivat gebräuchlich. Im Bereich der Informatik fällt zum Beispiel oft das Wort UNIX-Derivat und meint damit ein Betriebssystem, das aus einer bestimmten

UNIX-Version abgeleitet wurde und einen weitgehend ähnlichen Systemaufbau besitzt. Nun auch in MyCoRe gibt es (noch) den Begriff des Derivats. Ursprünglich ist seine Verwendung auch recht nah an der Wortbedeutung gesiedelt.

Bisher haben wir uns ausschließlich mit Metadaten befasst. So nützlich sie zur Katalogisierung und damit zur Recherche sind, sind sie doch nur Mittel zum Zweck. Es gilt schließlich in einem Dokumenten-Server, nicht Metadaten, zu verwalten, sondern Dokumente. Diese Dokumente liegen als Dateien vor, in ganz vielfältigen Formaten. Metadaten – MyCoRe-Objekte – sollen diese Dateien nur beschreiben und leichter auffindbar machen.

In MyCoRe versteht man *Derivate* als Objekte, die von einem Metadatenobjekt abgeleitet worden sind. Es ist zunächst schwierig das Konzept zu verstehen, wenn noch ausgeführt wird, das *Derivate* wiederum Objekte sind, die Metadaten enthalten. Im Unterschied zu MyCoRe-Objekten ist ihr Metadatenmodell festgelegt und damit nicht erweiterbar. Außerdem ist der Typ *derivate* in MyCoRe reserviert und darf somit nicht für MyCoRe-Objekte verwendet werden. Wie sieht die Verwendung von diesen *Derivaten* aus?

Beispiel: Ein Professor stellt für seine Studenten die Folien seiner Vorlesung bereit. Die Vorlesung selbst wird als MyCoRe-Objekt eingestellt. Die Folien existieren im Original im PowerPoint-Format, zusätzlich existiert noch eine druckbare Version im PDF-Format. Beide Versionen stellen den gleichen Inhalt dar, in unterschiedlichen Präsentationsformen. Man bezeichnet sie als *Derivate* der Vorlesung (MyCoRe-Objekt), weil die selbstdefinierten Metadaten (z.B. Autor, Titel, Vorlesungsdatum) dieses Objekts beide Dateien beschreiben. Es hebt also ihre Ähnlichkeit hervor.

Nun kann MyCoRe diesen semantischen Zusammenhang nicht sicherstellen, weshalb die ursprüngliche Bedeutung so sich nicht auf jedes MyCoRe-System übertragen lässt. Vielmehr versteht man im Betrieb unter einem *Derivat* nur noch eine Menge von Dateien, die einem, oder auch mehreren MyCoRe-Objekten zugeordnet sind. Die Notwendigkeit mehrere Dateien zu einer Gruppe zusammenzufassen, ergibt sich zum Beispiel beim HTML-Format. Dieses Format erlaubt es strukturierte Dokumente zu erstellen und Teile (zum Beispiel Bilder) aus externen Quellen nachzuladen. Was ist also ein *Derivat*?

Derivat

Ein *Derivat* ist eine Zusammenfassung von Dateien, mit einer Hauptdatei, die einem oder mehreren MyCoRe-Objekten zugeordnet sind.

Codebeispiel 3.3 XML-Darstellung eines MyCoRe-Derivats

```
<?xml version="1.0" encoding="utf-8"?>
<mycorederivate
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="datamodel-derivate.xsd"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
  ID="DiplArb_derivate_00000001"
  label="Derivat von DiplArb sample 1">
  <derivate>
    <linkmetas class="MCRMetaLinkID">
      <linkmeta xlink:type="locator"
        xlink:href="DiplArb_sample_00000001" />
    </linkmetas>
    <internals class="MCRMetaIFS">
      <internal
        sourcepath="vorlesung20050411"
        maindoc="vortragsfolien.pdf" />
    </internals>
  </derivate>
</service>
</mycorederivate>
```

Ein Derivat kann also nach dieser Definition zwar zu mehreren MyCoRe-Objekten gehören, jedoch nicht allein im System existieren.

Codebeispiel 3.3 zeigt ein Derivat in seiner XML-Form. Die Liste von zugeordneten Dokumenten befindet sich unter dem `<linkmetas>`-Tag. MyCoRe orientiert sich hier an dem XLINK-Standard (DeRose u. a. 2001). Das Beispiel-Derivat ist dem Dokument mit der ID `DiplArb_sample_00000001` zugeordnet, also dem bereits aus Codebeispiel 3.1 bekannten. Das Tag `<internal>` sagt dem MyCoRe-System, wo es die Dateien des Derivats findet (`sourcepath`-Attribut; hier relative Pfadangabe), während `maindoc` die in der Definition erwähnte Hauptdatei bestimmt. Wie in 3.2 folgt nach dieser kurzen Einführung ein Überblick über die Klassen, die MyCoRe verwendet, um Derivate zu repräsentieren.

Klasse: MCRDerivate

Aus 3.2.2 ist bereits bekannt, dass `MCRObject` und `MCRDerivate` von der abstrakten Klasse `MCRBase` (siehe Abb. 3.5) abgeleitet sind. Die Klasse `MCRDerivate` instanziiert die `MCRObjectDerivate`-Klasse als Feld `mcr_derivate`, in dem die Informationen innerhalb des `<derivate>`-Tags verwaltet werden. Der Ablauf bei der Instanziierung des `MCRDerivate`-Objekts orientiert sich an dem des `MCRObject`-Objekts. Das Wurzel-Element

(hier: `mycorederivate`) wird auf die gleiche Weise ausgewertet, da sich der Aufbau, abgesehen vom Tag-Namen natürlich, nicht vom `<mycoreobject>` unterscheidet. Es ist wieder ein Service-Bereich vorgesehen, der wie zuvor in 3.2.2 verwendet wird. **Tabelle 3.3** zeigt eine Übersicht der soeben erwähnten Objekte.

Datenfeld	Typ	Basisklasse
<code>mcr_id</code>	<code>MCRObjectID</code>	<code>MCRBase</code>
<code>mcr_label</code>	<code>java.lang.String</code>	<code>MCRBase</code>
<code>mcr_schema</code>	<code>java.lang.String</code>	<code>MCRBase</code>
<code>mcr_derivate</code>	<code>MCRObjectDerivate</code>	<code>MCRDerivate</code>
<code>mcr_service</code>	<code>MCRObjectService</code>	<code>MCRBase</code>

Tabelle 3.3: Bestandteile des MCRDerivate

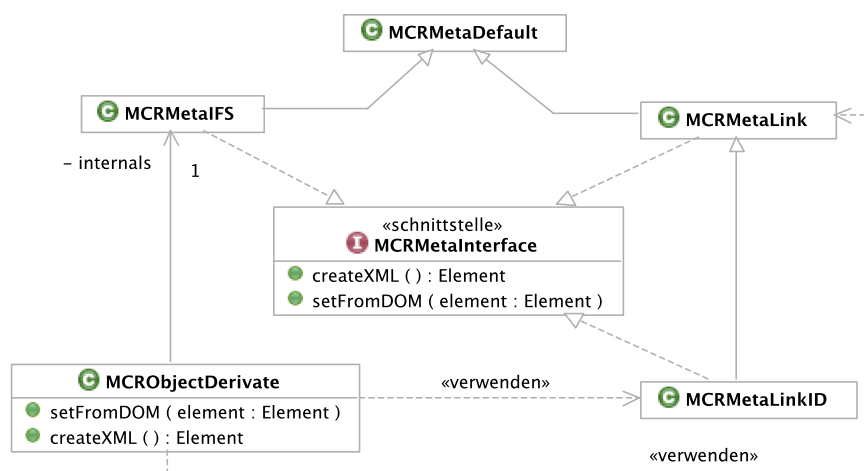


Abbildung 3.4: Klassenhierarchie für das `<derive>`-Element

Klasse: `MCRObjectDerivate`

Der gesamte Inhalt des `<derive>`-Elements wird von der Klasse `MCRObjectDerivate` verwaltet. Das Codebeispiel 3.3 lässt vermuten, dass für `<linkmetas>` und `<internals>` wieder unterschiedliche Klassen für die Auswertung der `<linkmeta>`- und `<internal>`-Tags verwendet werden können. Jedoch wertet MyCoRe das Attribut `class` nicht aus, so dass in jedem Fall die Klassen `MCRMetaLinkID` bzw. `MCRMetaIFS` verwendet werden. Die Software-Architektur (siehe Abb. 3.4) ist diesbezüglich auch an anderen Stellen fest verdrahtet, so sind die Rückgabewerte für die Methoden

`getLinkMeta(int)` und `getInternals()` jeweils `MCRMetaLinkID` und `MCRMetaIFS`. Eine mögliche Flexibilisierung der Klasse würde also größere Änderungen der API¹⁰ nach sich ziehen.

Klasse: `MCRMetaIFS`

Die Klasse `MCRMetaIFS` ist, wie ihr Name vermuten lässt, eine Klasse, die die `MCRMetaElement`-Schnittstelle implementiert, könnte also durchaus in einem `<mycoreobject>` auftauchen. Zusätzliche Funktionalitäten, wie das Abspeichern der Dateien wird von ihr nicht angeboten. Über die Methode `createXML()` wird, wie schon beim `MCRObject`, aus der Klasse die zugehörige XML-Repräsentation erzeugt.

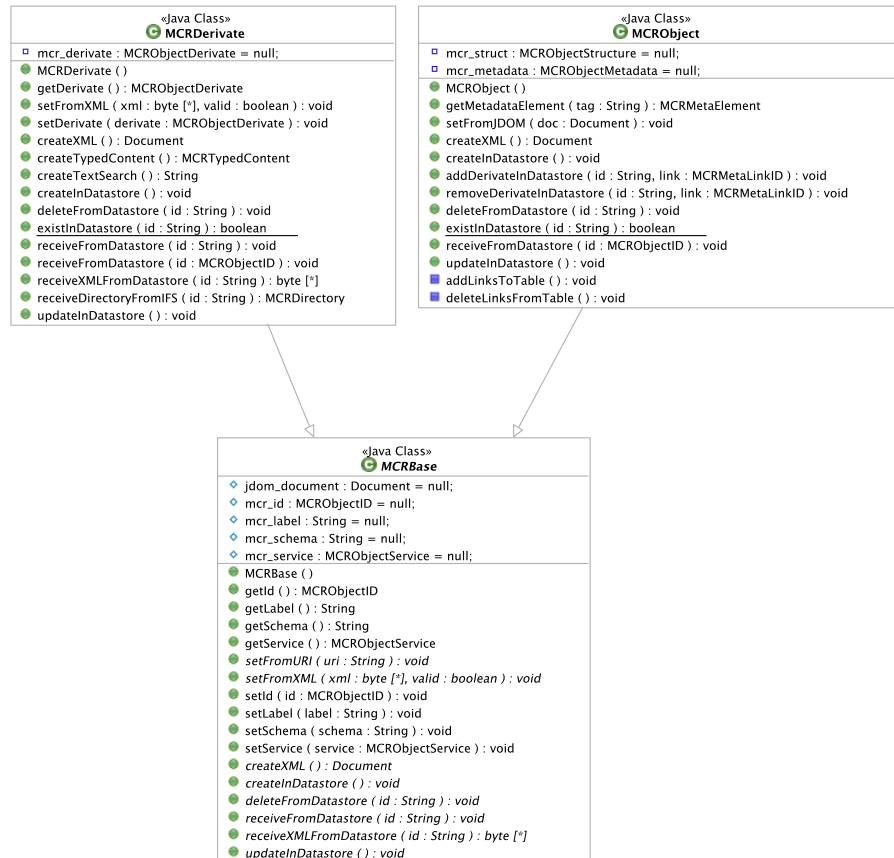


Abbildung 3.5: Klassenhierarchie `MCRBase`, `MCRObject`, `MCRDerivate`

¹⁰Application Programming Interface

3.4 Die MCRClassification

Der Typ `class` ist der Zweite von MyCoRe reservierte Typ, der nicht zur freien Verwendung ist. Klassifikationen sind gegliedert in Kategorien, die jeweils andere Kategorien enthalten können. Diese kann man sich wie Schachteln vorstellen, in die Dokumente abgelegt werden können. Dabei kann ein MyCoRe-Objekt verschiedenen Klassifikationen gleichzeitig zugeordnet werden, aber in der Regel immer nur jeweils einer Kategorie. **Codebeispiel 3.4** zeigt eine einfache und recht kleine Klassifikation.

Codebeispiel 3.4 XML-Darstellung einer MyCoRe-Klassifikation

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mycoreclass
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="MCRClassification.xsd"
  ID="DiplArb_class_00000001" >
  <label
    xml:lang="de"
    text="DiplArb Test-Klassifikation"
    description="ein einfaches Beispiel einer Klassifikation"/>
  <categories>
    <category ID="PROF0000">
      <label xml:lang="de" text="universitätsangehörig"/>
      <label xml:lang="en" text="university affiliated"/>
      <category ID="PROF0000.Jena">
        <label xml:lang="de" text="FSU-Jena"/>
        <label xml:lang="en" text="University of Jena"/>
      </category>
      <category ID="PROF0000.other">
        <label xml:lang="de" text="andere Universitäten"/>
        <label xml:lang="en" text="other universities"/>
      </category>
    </category>
    <category ID="PROF0001">
      <label xml:lang="de" text="freie Wirtschaft"/>
      <label xml:lang="en" text="private enterprise"/>
    </category>
    <category ID="PROF0002">
      <label xml:lang="de" text="arbeitssuchend"/>
      <label xml:lang="en" text="out of engagement"/>
    </category>
  </categories>
</mycoreclass>
```

Das Beispiel enthält auch eine Verschachtelung, so sind die Kategorien `PROF0000.Jena` und `PROF0000.other` Kategorien, die in der Kategorie `PROF0000` enthalten sind. Das bedeutet, dass jedes MyCoRe-Objekt, das in `PROF0000.Jena` enthalten ist, auch in `PROF0000` enthalten ist. Umgekehrt

gilt dies natürlich nicht. Neben der ID kann man noch sogenannte Labels angeben. Sie beschreiben, in der jeweils angegebenen Sprache, die Kategorie, für den Fall dass die ID, wie hier PROF0000, nicht aussagekräftig genug ist.

3.5 Suche in MyCore

Bereits in (Lützenkirchen u. a. 2004, Kapitel 3.1) wird das Query-Modell von MyCoRe beschrieben, das in (Scheffler 2004) um eine Volltextsuche in den Dateien erweitert wurde. In diesem Abschnitt wird deshalb nur zum leichteren Verständnis kurz auf das Query-Modell eingegangen, um dann die Implementierung besser verstehen zu können.

Bei der Architektur des QueryModells kam es den Entwicklern darauf an, sich an vorhandene Standards anzulehnen. Im Fall der Suche entschied man sich daher, die XPATH-Syntax (Clark u. DeRose 1999) für Suchanfragen zu verwenden und sie zu erweitern, wo dies notwendig war. Zur Zeit existieren in MyCoRe drei feste XML-Dokument-Wurzeln: `mycoreobject` für Metadaten, `mycorederivate` für Derivate und `mycoreclass` für Klassifikationen. Diese stellen Objektklassen dar, die suchbar sind. In Abschnitt 3.2 haben wir bereits ein kurzes Beispiel eingeführt. Da die Suchanfragen für diese Klassen sich im Aufbau nicht groß unterscheiden, wird das Beispiel hier fortgeführt.

Die Suchanfrage `/mycoreobject[@ID="MyCoRe_sample_1"]` liefert das **Codebeispiel 3.5**. Man sieht, dass der komplette Code aus Codebeispiel 3.1 darin enthalten ist, eingebettet in ein `<mycoreresults>`-Dokument, das die einzelnen Ergebnisse (in dem Beispiel genau das eine) in einem `<mycoreresult>`-Container zusammen fasst.

Die Suchanfrage selbst ist schnell geklärt. Sie zielt auf *alle* Dokumente ab, deren ID - das Attribut "ID" von `<mycoreobject>` ist gemeint - den Wert "MyCoRe_sample_1" haben. Da MyCoRe sicherstellt, dass eben diese ID eindeutig ist, liefern Anfragen wie diese entweder kein oder aber genau ein Ergebnis. Eine etwas kompliziertere Anfrage könnte etwa so aussehen:

```
/mycoreobject[metadata/creatorlinks/creatorlink/
@xlink:href="MyCoRe_person_00000001" and doctext()
contains("All your base" -foo)]
```

Diese Anfrage zeigt die Kombination mehrerer Suchanfragen. Zum einen die nach allen Dokumenten, deren Autor die ID `MyCoRe_person_00000001` besitzt, zum anderen die nach allen Dokumenten, in deren Dateien die Wortgruppe „All your base“ enthält, aber nicht das Wort „foo“. Das ‚and‘ in der Anfrage stellt sicher, dass nur Ergebnisse zurück geliefert werden, die von beiden *Subqueries* erzeugt wurden.

Codebeispiel 3.5 XML-Darstellung eines Suchergebnisses

```

<?xml version="1.0" encoding="utf-8"?>
<mcrresults>
  <mcrresult>
    <mycoreobject
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="datamodel-sample.xsd"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      ID="DiplArb_sample_1"
      label="Beispiel Objekt 1">
      <structure />
      <metadata>
        <titles class="MCRMetalangText" textsearch="true">
          <title xml:lang="de">
            Nur ein Beispielobjekt
          </title>
          <title xml:lang="en">
            Just a sample object
          </title>
        </titles>
        <creatorlinks class="MCRMetalinkID">
          <creatorlink xlink:type="locator"
            xlink:href="Dipl_person_00000001"
            xlink:label="Author"
            xlink:title="Thomas Scheffler"/>
        </creatorlinks>
      </metadata>
    </mycoreobject>
  </mcrresult>
</mcrresults>

```

Das Datenmodell für `mycoreobject` ist variabel konfigurierbar und kann in unterschiedlicher Form mehrfach vorkommen, z.B. Dokumente vom Typ „sample“ und „person“. Diese Flexibilität macht es erforderlich zusätzlich zu der XPATH-Anfrage noch die Dokument-Typen mitzuliefern. Dies kann eine Liste der Form `Typ1, Typ2, Typ3, ...` sein, wenn alle Typen alle Elemente der XPATH-Anfrage enthalten. Zum Beispiel sei angenommen, dass die Typen „Brief“ und „Buch“ das Element `metadata/creatorlinks/creatorlink` enthalten, um einen Autor zu kennzeichnen. Dann kann die Suche über alle Briefe und Bücher nach denen eines bestimmten Autors in einer Anfrage definiert werden.

3.6 Persistenz

3.6.1 Persistenz von `<mycoreobject>`

Die Spezifika des `<mycoreobject>` sind das flexible Metadatenmodell, die mögliche Vererbungshierarchie und Objektverlinkung. In diesem Punkt wird die Datenintegrität durch MyCoRe nicht gewährleistet. Das bedeutet, dass man ein Objekt löschen kann, z.B. ein Person-Objekt, das einem anderen zugeordnet ist, z.B. als Autor innerhalb eines Buch-Objekts. Der Löschvorgang stößt in MyCoRe 1.0 keine Aktionen an, die diese Art der referentiellen Integrität sicherstellt. Es ist also möglich, dass der Autor des Buches zwar referenziert ist, aber nicht mehr existiert. Schlimmer noch, MyCoRe verhindert zukünftig nicht, dass ein logisch anderes Person-Objekt mit der gleichen ID eingestellt wird, was defacto dem Buch einen falschen Autor „unterschiebt“.

Zukünftig muss dieser Zustand in MyCoRe überarbeitet werden, so dass dieser nicht mehr auftreten kann.

Einen Spezialfall stellt die referentielle Integrität in einer Vererbungshierarchie dar. MyCoRe stellt hier sehr wohl sicher, dass Waisenkinder ausgeschlossen sind, indem Löschoperationen grundsätzlich kaskadierend erfolgen. In dem ersten Teil wird nun beschrieben, wie ein `<mycoreobject>` persistent abgelegt wird. Ein paar der hier beschriebenen Konzepte tauchen im Abschnitt 3.6.2 zum `<mycorederivate>` auf, so dass dort auf eine wiederholte ausführliche Erläuterung verzichtet wird.

Über die Methode `createXML()` wird, wie in 3.2 beschrieben, rekursiv die vollständige XML-Repräsentation in Form eines JDOM-Objektes erstellt. Über dieses JDOM-Dokument ist man in der Lage, den XML-Code in verschiedene Ausgabeformate zu überführen. Möglich sind auf diesem Wege z.B. eine Ausgabe über einen `OutputStream` oder SAX¹¹-Events.

Sicherung von `MCRObject`

Um ein `MCRObject` zu sichern, wird die Methode `createInDatastore()` aufgerufen. **Abbildung 3.6** stellt den internen Ablauf vereinfacht dar. Es fehlen Operationen, die Ausnahmen (Exceptions) hervorrufen, um das Abspeichern fehlerhaften Codes zu verhindern. Des weiteren sind Code-Teile, die Vererbungshierarchien betreffen, ausgeblendet, sie liegen in den beiden `opt`-Blöcken.

Zunächst wird im `<service>`-Bereich das aktuelle Datum als Erstellungsdatum und Datum der letzten Änderung gesetzt. Der nun folgende Aufruf von `createXML()` dient laut Code-Kommentar der Validierung. Neben der

¹¹<http://www.saxproject.org/>

Validierung der MyCoRe-ID, hier kann eine Ausnahme auftreten, wird noch die komplette JDOM-Sicht erzeugt. Das entstehende JDOM-Dokument wird aber zu diesem Zeitpunkt nicht vorrätig gehalten. Es folgt der Block in dem die Beziehung zum Vaterobjekt überprüft wird. Es wird sichergestellt, dass falls ein Vater angegeben ist (`parentID!=null`), er auch existiert. Nachdem das Vaterobjekt mittels `receiveFromDatastore()` geladen wird, werden vererbte Metadaten dem aktuellen Objekt übertragen. Auch an dieser Stelle wird das `MCRObject` des Vaters für die spätere Verwendung nicht vorrätig gehalten.

Für den `create()`-Aufruf des `MCRXMLTableManager` wird nun nochmals die `createXML()` Methode aufgerufen, weil das JDOM-Dokument des ersten Aufrufs nicht aufgehoben wurde. Dieser Aufwand müsste nur einmal betrieben werden. Es folgt eine weitere Speicherung über das `MCRObjectSearchStoreInterface` und dessen `create()` Methode, der sich das Objekt selbst übergibt.

Es ist kein offensichtlicher Grund erkennbar, wieso an dieser Stelle das Objekt an zwei verschiedenen Stellen vorgehalten werden muss. Dies schafft unnötige Redundanz, macht zusätzlichen Code erforderlich, der gepflegt werden muss und kostet Zeit während der Ausführung.

Der Aufruf von `deleteLinksFromTable()` und `addLinksToTable()` dient der Aktualisierung aller Referenzen des aktuellen Objekts. MyCoRe sichert Referenzen zu anderen Objekten und Klassifikationen in einer relationalen Datenbank. Über diese Daten wäre eine referentielle Integrität, die wie oben beschrieben nicht existiert, zu realisieren.

Der letzte Parent-Block dient dazu – nach der nun erfolgreichen Speicherung des Kindes – das Kind im `<structure>`-Teil des Vaters einzufügen. Schlägt dieser Aktualisierungsversuch fehl, wird das Kind wieder aus der Datenhaltungsschicht über `deleteFromDatastore()` entfernt. Es wird an dieser Stelle nicht ermöglicht, diese Ausnahme abzufangen und von der aufrufenden Methode zu verarbeiten. Es empfiehlt sich daher, nach „erfolgreicher“ Speicherung, mittels `existInDatastore()` die Existenz selbst sicherzustellen.

Laden von `MCRObject`

Abbildung 3.7 zeigt den Ablauf beim Laden einer `MCRObject`-Instanz. Über den `MCRXMLTableManager` wird als Byte-Array der XML-Code geholt. Wie in 3.2 bereits beschrieben, wird nun, falls das Byte-Array vorhanden ist (`xml!=null`), durch Aufruf von `setFromXML()` das `MCRObject` mit Daten gefüllt. Falls das Auslesen des `MCRXMLTableManager` erfolglos war, wird eine `MCRPersistenceException` geworfen, welche die aufrufenden Methoden abfangen müssen. An dieser Stelle wird der eigentliche

Verwendungszweck des `MCRXMLTableManager` deutlich, er stellt eine Möglichkeit bereit, über die ID des Objekts dieses abzuspeichern und auszulesen (**Abb. 3.8**). Die Entwickler gingen wahrscheinlich davon aus, dass eine Suche nach der ID über das `MCRObjectSearchStoreInterface` zu ineffizient ist. Das `MCRObjectSearchStoreInterface` erlaubt nämlich nur Schreibzugriff, wie in **Abb. 3.9** zu erkennen ist. Dabei ist es auf einfache Weise möglich beide Schnittstellen zusammenzuführen. Sollte eine natürliche Implementierung des `MCRObjectSearchStoreInterface` tatsächlich Performance-Probleme beim Zugriff über IDs haben, so kann der Programmierer dann tatsächlich auf eine datenbankgestützte XML-Code-Beschaffung ausweichen. Vom Gesichtspunkt der Software-Architektur macht eine „Write-Only“-Schnittstelle keinen Sinn. In diesem besonderen Fall ist es zudem so, dass keine Methode einen Rückgabewert liefert. Jegliche Rückkopplung zur aufrufenden Methode erfolgt über Exceptions. Die Schnittstelle verhält sich in diesem Zusammenhang wie ein schwarzes Loch, das alles schluckt, aber nichts mehr herausgibt. Mit einem Zusammenführen von `MCRXMLTableInterface` und `MCRObjectSearchStoreInterface` würde man sich die erzwungene zweifache Speicherung sparen können.

Löschen von `MCRObject`

Das Löschen eines `MCRObject`s ist in **Abb. 3.10** dargestellt. Das Sequenzdiagramm ist um Logging und Ausnahmebehandlung erleichtert und trotzdem noch recht umfangreich. Ursächlich ist für dieses Verhalten bestimmt anzubringen, dass MyCoRe sicherstellen will, dass Waisenkinder sowohl bei MyCoRe-Objekten wie bei MyCoRe-Derivaten nicht auftreten. Für alle Kinder (MyCoRe-Objekte und Derivate) wird rekursiv das Löschen angestoßen, wobei davor jeweils die Daten ausgelesen werden müssen und die entsprechenden Objekte initialisiert werden müssen. Ein Aufwand, den eine mögliche bessere Implementierung in der Persistenz-Abbildungsschicht verhindern könnte. Nach dem erfolgreichen Löschen wird das eventuelle Vater-Objekt geladen und das Objekt als Kind aus dem Strukturbereich ausgetragen. Es folgt das Löschen aus dem `MCRObjectSearchStoreInterface`, das Entfernen externer Referenzen mittels `deleteLinksFromTable()` und das Löschen beim `MCRXMLTableManager`.

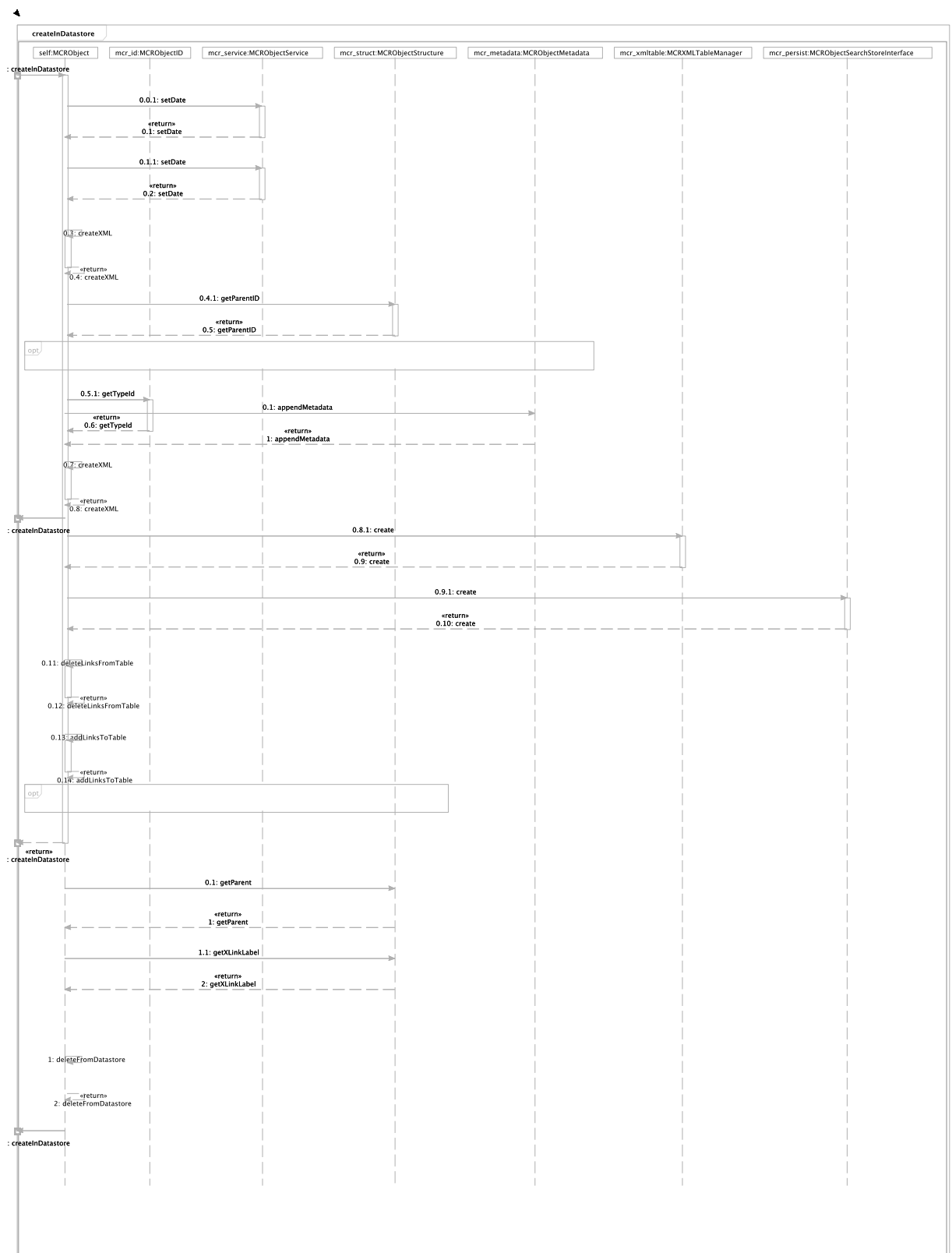


Abbildung 3.6: createInDatastore-Methode von MCRObject

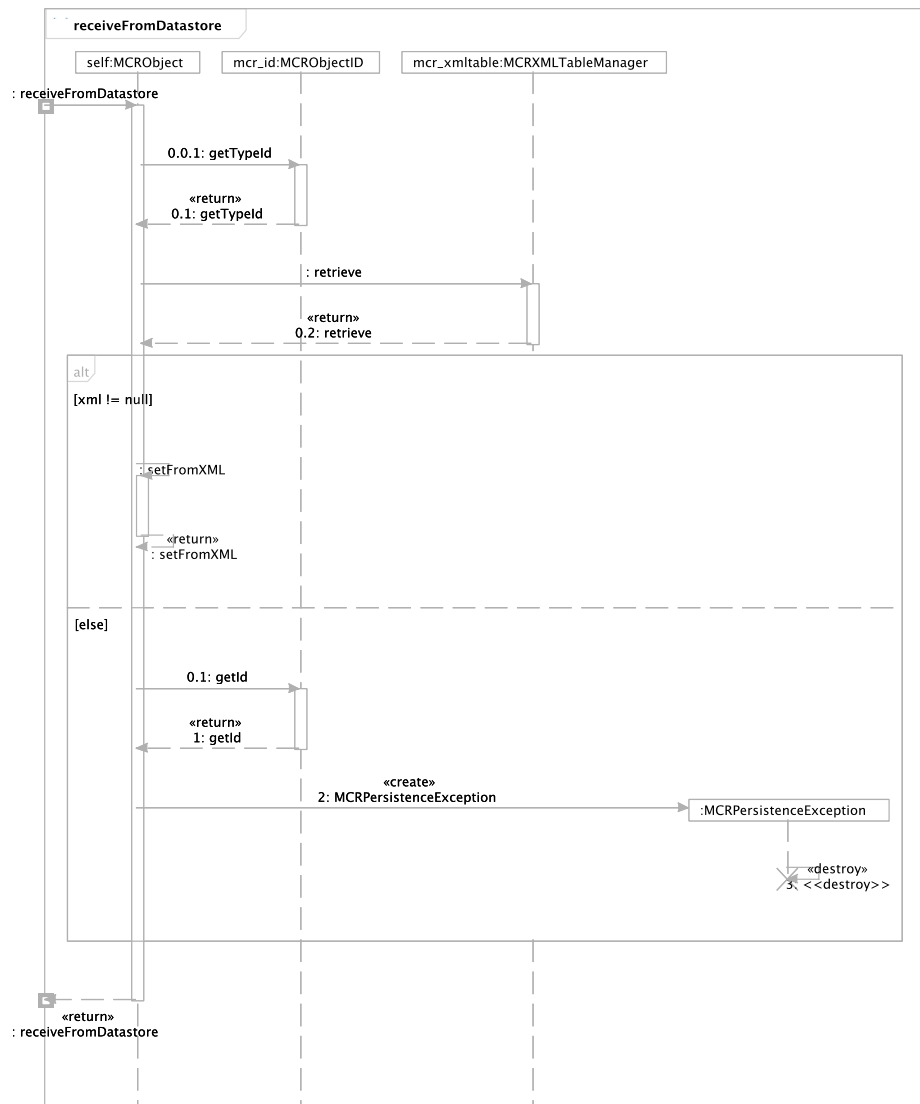


Abbildung 3.7: receiveFromDatastore-Methode von MCRObject

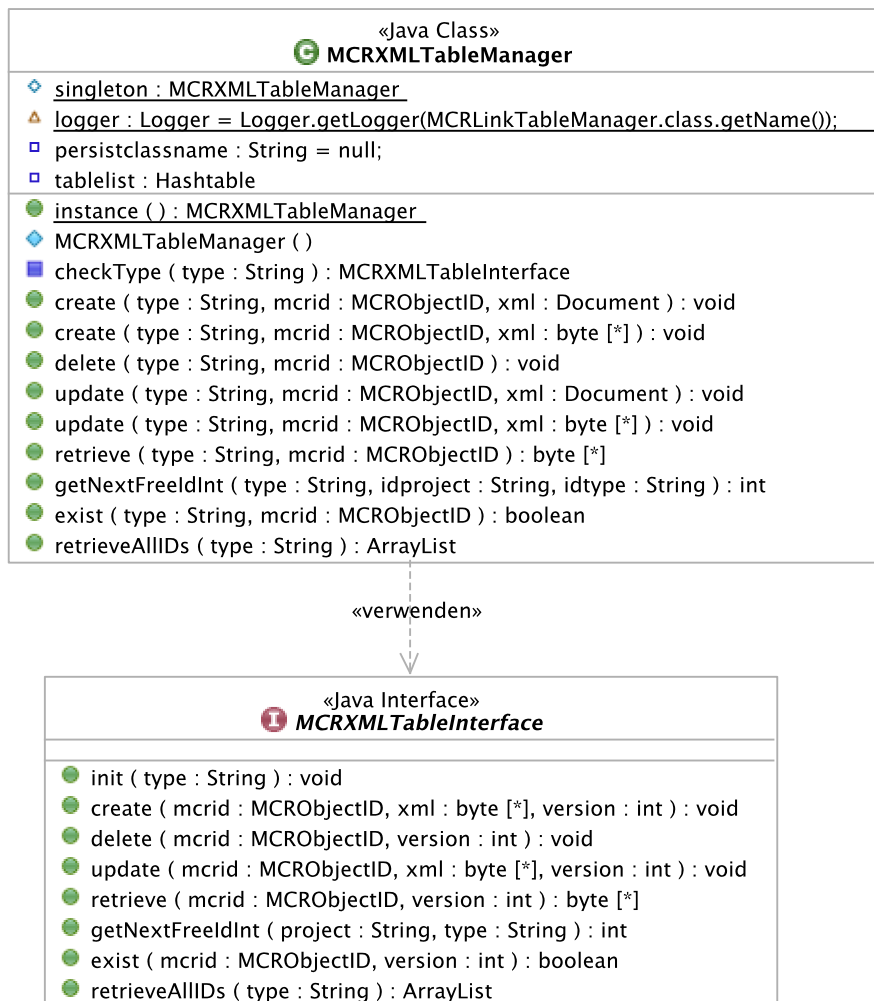


Abbildung 3.8: Klassendiagramm: MCRXMLTableManager und MCRXMLTableInterface

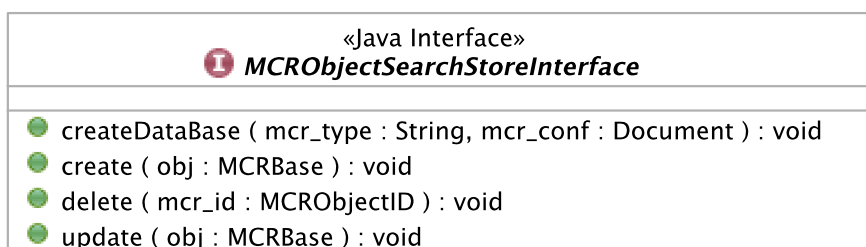


Abbildung 3.9: Klassendiagramm: MCRObjektSearchStoreInterface

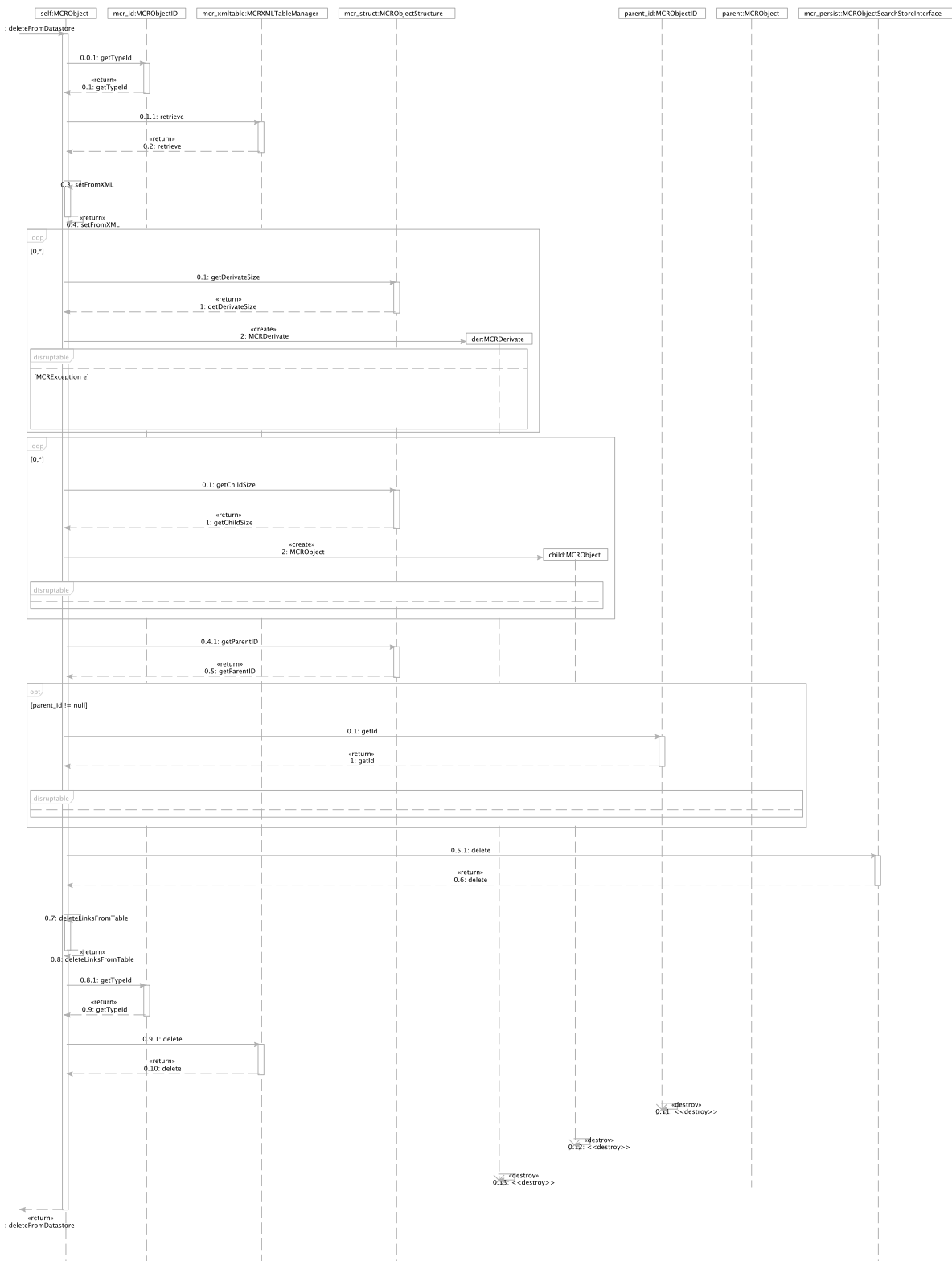


Abbildung 3.10: deleteFromDatastore-Methode von MCRObject

3.6.2 Persistenz von `<mycorederive>`

Speicherung der Derivat-Metadaten

Bereits in 3.3 wurde die Gemeinsamkeit von `MCRObject` und `MCRDerivate` herausgestellt. Entsprechend sind die Abläufe ähnlich, was die Persistenz der Derivate betrifft. Im nächsten Abschnitt wird die Speicherung der Dateien im *Internal File System* erläutert, während sich dieser Abschnitt mit der Speicherung der Metadaten beschäftigt.

Diese beiden Aspekte der Speicherung erfolgen bei MyCoRe über die `createInDatastore()`-Methode, wie schon zuvor bei `MCRObject`. In dieser Methode wird weitgehend sichergestellt, dass Speichern von Dateien und Metadaten nur zusammen erfolgt oder gar nicht. Folgender Ablauf stellt dies kurz dar:

1. Sichern der Metadaten mittels `MCRXMLTableManager` analog zu `MCRObject`
2. Importieren aller Derivat-Dateien ins IFS
 - (a) Aufruf der `MCRFileImportExport.importFiles(File local, String ownerId)` liefert `MCRDirectory`-Instanz (Dateien sind gesichert)
 - (b) ID von `MCRDirectory` wird in `<internals>` des Derivats gesichert

Speicherung über `MCRObjectSearchStoreInterface` analog zu `MCRObject`

- 3.
4. Sicherung `MCRMetaLinkID` zu zugehörigen `MCRObjecten`

Scheitert jeweils ein Punkt (1, 2, 3 oder 4), werden alle vorhergehenden rückgängig gemacht. Dies entspricht der Atomaritätseigenschaft von Transaktionen (Häder u. Reuter 1983). Was dabei auffällt, ist, dass zwischen getrennter Speicherung der Derivat-Metadaten über `MCRXMLTableManager` und `MCRObjectSearchStoreInterface` die Metadaten verändert werden (Punkt 2). Bei der zukünftigen Entwicklung eigener Applikationen oder des MyCoRe-Kerns muss man diesen Umstand berücksichtigen. Es gelten ferner die Anmerkungen, die diesbezüglich schon bei `MCRObject` gemacht worden sind, hier im gleichen Zuge.

Das Löschen eines Derivats gestaltet sich auch sehr ähnlich zu dem des `MCRObject`s, weshalb an dieser Stelle dieser Aspekt nicht ausgeführt wird. Es

wird lediglich zusätzlich das zugehörige `MCRDirectory` gelöscht und somit sichergestellt, dass gesicherte Dateien aus dem IFS entfernt werden.

IFS – Internal File System

In diesem Abschnitt wird das MyCoRe-eigene *Internal File System* beschrieben, wie es schon in (Lützenkirchen 2003) dargestellt wurde.

Bisher wurde die Persistenz von Metadaten gezeigt. MyCoRe verwaltet jedoch nicht nur die Metadaten zu Dateien, sondern diese auch selbst. Dabei wurden in MyCoRe typische Operationen eines Dateisystems implementiert, mit denen transparent auf gespeicherte Dateien zugegriffen werden können. MyCoRe arbeitet dabei auch mit den Begriffen Verzeichnis und Datei, diese werden `MCRDirectory` und `MCRFile` genannt.

Das Internal File System – kurz IFS – ist jedoch weit mächtiger als ein normales Dateisystem. Es bietet nicht nur eine Schnittstelle, über die mittels eines Browsers auf alle Dateien zugegriffen werden kann. Es unterstützt auch die Volltextindizierung (Scheffler 2004) der abgespeicherten Dateien und eine Reihe von Export- und Import-Funktionen.

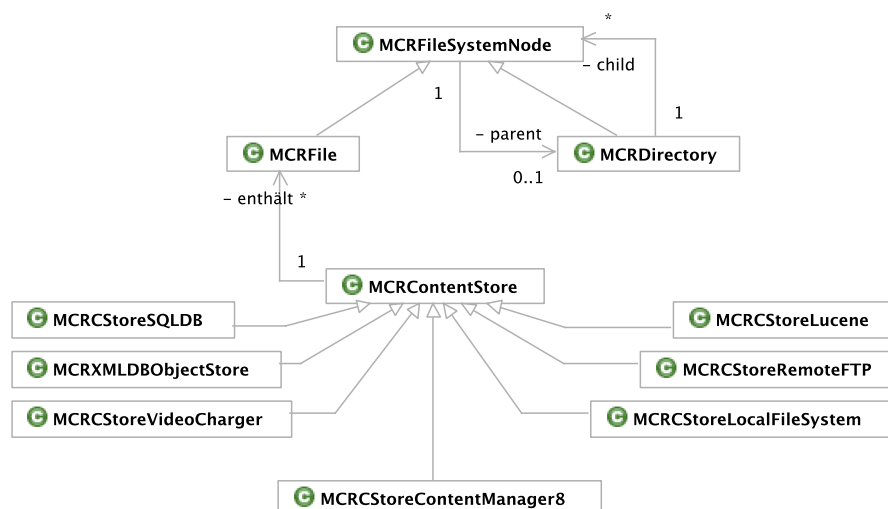


Abbildung 3.11: Klassendiagramm: IFS – Internal File System

In **Abbildung 3.11** ist die Architektur von MyCoRes IFS dargestellt. Aus ihr wird deutlich, dass Dateien in verschiedenen ContentStores¹² liegen können, logisch aber zu einem Verzeichnis gehören. Ganz den bekannten Dateisystemen

¹²Dateispeicher mit möglichen Spezialisierungen für bestimmte Dateitypen, z.B. Volltextindizierung

nachempfunden, verwaltet auch MyCoRe Verzeichnisse in Verzeichnissen (Unterverzeichnisse). Dies ist zum Beispiel bei HTML-Präsentationen von Nutzen, bei denen selten alle Dateien innerhalb eines Verzeichnisses liegen.

Für das IFS gibt es eine relevante Schnittstelle und eine abstrakte Klasse, die für die Sicherung der Dateien verantwortlich ist. **Abbildung 3.12** zeigt eine Auswahl von Klassen des IFS. Das `MCRFileMetadataStore`-Interface ist das Kernstück. In einem MyCoRe-System läuft genau eine Implementierung dieser Klasse, ein „Mischbetrieb“, wie er sonst in MyCoRe möglich ist, funktioniert hier nicht, er ist zudem nicht sinnvoll. Die Implementierung dieser Schnittstelle kümmert sich um nichts geringeres, als die Metadaten zu den Dateien zu speichern. Sie ist die Anlaufstelle, um auf Dateien zu zugreifen, denn die Implementierung bildet auch die Verzeichnisstruktur ab. Für MyCoRe existiert zur Zeit eine Implementierung dieser Schnittstelle auf relationaler Datenbankebene. *IFS erfordert DBMS in MyCoRe* Für Single-Backend-Systeme, wie sie vielleicht beim Content Manager V8 sinnvoll sein könnten, kann eine eigene Implementierung die doppelte Buchführung unnötig machen, wenn geeignete ContentStore-Backends diese Funktionalität von Haus aus bieten. Drei weitere Elemente aus **Abbildung 3.11** (Node, ContentStore, File) sind ebenfalls im Klassendiagramm aufgeführt.

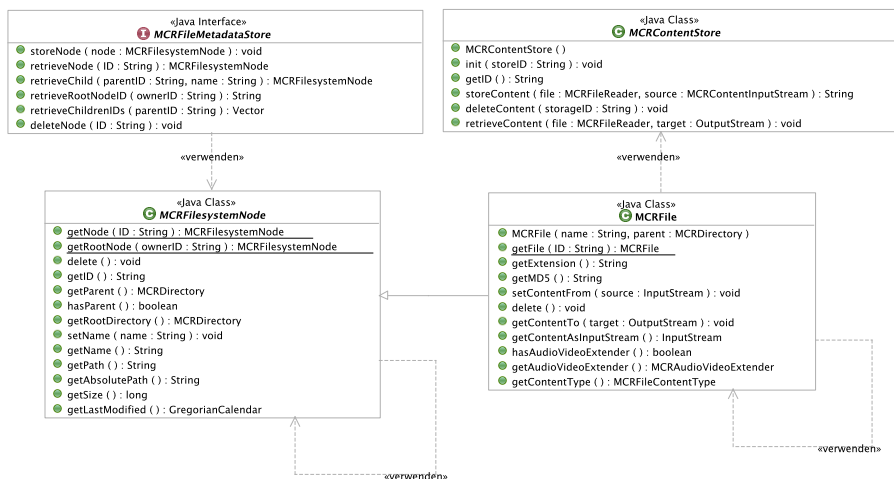


Abbildung 3.12: Klassendiagramm: `MCRFileMetadataStore`, `MCRContentStore`, `MCRFilesystemNode`, `MCRFile`

Die Zuordnung von `MCRFile` zu einem `MCRContentStore` erfolgt über so genannte *ContentStoreSelectionRules*. In diesen Regeln kann man einzelne Dateitypen (zum Beispiel: PDF, Word97, JPEG) bestimmten ContentStores zu-

ordnen. MyCoRe beachtet bei der Zuordnung von Datei zu Dateityp sowohl Dateieindung, als auch so genannte Magic-Bytes¹³.

Es soll ferner nicht unterschlagen werden, dass ein `MCRFile` einem so genannten `MCRAudioVideoExtender` zugeordnet werden kann. Dessen Implementierungen (für Real Helix Server und IBM VideoCharger) erlauben das Streaming dieser Dateien und stellen eine Reihe zusätzlicher, automatisch erfasster Metadaten bereit (Bitrate, Länge, Codec etc.). Die genaue Funktionsweise wird aus Gründen des Aufwandes und Spezialisierung hier nicht weiter erläutert, weil sie an den prinzipiellen Abläufen im IFS nichts ändert.

3.6.3 Persistenz von `<mycoreclass>`

Sicherung von `MCRClassification`

Die Klassifikationen in MyCoRe sind ein gänzlich anderes Konzept und nicht zu vergleichen mit `MCRObject` und `MCRDerivate`. Im Normalfall werden Klassifikationen vor den anderen MyCoRe-Objekten eingestellt – spätestens jedoch, bevor `MCRObject`-Instanzen in Kategorien einer bestimmte Klassifikation einsortiert werden.

Die Methode `setFromJDOM(Document jdom)` (**Codebeispiel 3.6**) ist in der untersuchten Version recht umständlich, komplex und schwer verständlich implementiert. Die Komplexität würde ich auf ein Ablaufdiagramm übertragen und das Verständnis für die Methode nicht erhöhen. Es wird im Fall von `MCRClassification` (**Abb. 3.13**) verzichtet, ein solches Ablaufdiagramm anzugeben, welches die Funktionsweise der entscheidenden Methode beschreibt. Eine Klassifikation kann zum Beispiel mehrere Labels, welche die Klassifikation beschreiben, enthalten, zusätzlich beinhaltet es genau EIN `<categories>`-Tag. Trotzdem wird über alle Kind-Elemente von `<mcrclass>` iteriert (Zeile 76-90). Abhängig von Tagnamen (`<label>` oder nicht) wird die Information in einem `MCRClassificationItem`-Object abgespeichert oder das Tag (JDOM-Element) geklont als `categories` in der Klasse gespeichert. Alternativ über die `<label>`-Elemente zu iterieren und das `<categories>`-Element getrennt davon zu verarbeiten, würde nicht nur unnötige Überprüfungen nach Tag-Namen reduzieren, sondern den Code zudem verständlicher.

Kategorien können selbst wiederum Kategorien enthalten. Zu jeder Kategorie kann noch eine URL angegeben werden, falls man damit eine Kategorie näher beschreiben möchte. Hier wird, wie schon bei `<categories>`, über alle Elemente iteriert, statt Labels und Kategorien getrennt zu behandeln und somit unnötige IF-THEN-ELSE-Blöcke einzusparen. Die Zeilen 93-143 sind recht

¹³Markante Bytes an bestimmten Positionen (z.B. Datei-Header) innerhalb einer Datei.

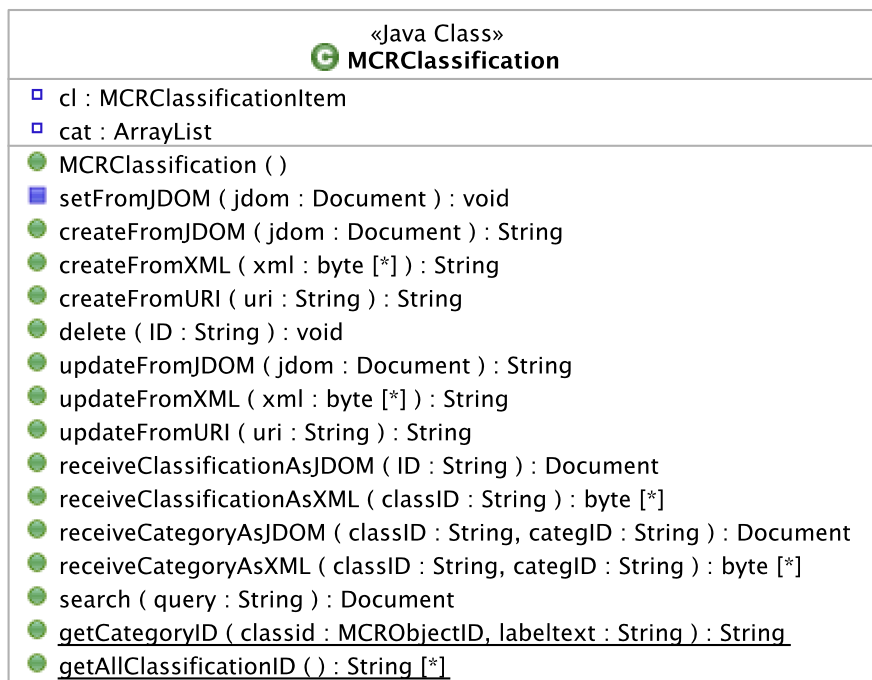


Abbildung 3.13: Klassendiagramm: MCRClassification

kompliziert. Hier wird die Klassifikationshierarchie abgearbeitet und alle Kategorien (Hierarchie) in eine flache Liste (`cat`; Zeile 134) von `MCRCategoryItem`-Instanzen gepackt.

Die Klasse `MCRCategoryItem` stellt Informationen bereit, welches `MCRClassificationObject` (`MCRClassificationItem` oder `MCRCategoryItem`; siehe **Abb. 3.14**) das übergeordnete Element in der Kategorie-Hierarchie ist. Somit lässt sich aus der flachen Liste `cat` durchaus wieder die Hierarchie rekonstruieren.

Hat `setFromJDOM()` erfolgreich seine Arbeit abgeschlossen, wird die Methode `create()` von `cl` (siehe Zeile 75) aufgerufen und danach jeweils von allen Kategorien (`MCRCategoryItem` in Liste `cat`).

Mittels `MCRClassificationManager` erfolgt diese Speicherung, der dafür eine Implementierung des `MCRClassificationInterface` benutzt (**Abb. 3.15**). Nachdem die Klassifikation nun in `MyCoRe` gespeichert ist, können `MCRObject`-Instanzen den Kategorien zugewiesen werden. Das Prinzip ist hierbei das gleiche, wie die Zuordnung von zwei `MCRObject`-Instanzen zueinander (`MCRMetaLinkID`; siehe Abschnitt 3.2), jedoch wird für Kategorien der spezielle Metadatentyp `MCRMetaClassification` verwendet (siehe Persistenz von `MCRObject` in Abschnitt 3.6.1). `MCRClassificationObject`



Abbildung 3.14: Klassendiagramm: MCRClassificationObject

stellt zusätzlich noch die Methode `countDocLinks()` bereit, mit der man die Anzahl der Dokumente bestimmen kann, die einer Kategorie zugeordnet sind.

Löschen von MCRClassification

Das Löschen einer MCRClassification gestaltet sich recht unspektakulär – verglichen mit der des MCRObject in 3.6.1. Über das Interface MCRClassificationInterface sind Löschoperationen sowohl für Klassifikationen (`deleteClassificationItem(String ID)`) als auch für Kategorien (`deleteCategoryItem(String CLID, String ID)`) definiert. Diese Definition ist aber recht vage gefasst und stellt keine weiteren Auflagen, wie dieses Löschen zu erfolgen hat:

„The method remove[s] a MCRClassificationItem from the datastore.“

(MCRClassificationInterface.java, Version: 1.7)

Was passiert mit Objekten, die Kategorien/Klassifikationen zugeordnet sind, welche nicht mehr existieren? Diese Frage wird implizit offengelassen. Da zudem keine Ausnahmen laut Interface abgefangen werden müssen, kann das vorzeitige Löschen von Klassifikationen zu einem inkonsistenten System führen. An dieser Stelle ist es notwendig die Schnittstellenspezifikation weiter zu konkretisieren, um diesem Fehlverhalten besser vorbeugen zu können.



Abbildung 3.15: Klassendiagramm: MCRClassificationManager

Codebeispiel 3.6 `setFromJDOM(Document jdom)-Methode` von `MCRClassification`

```

70 private final void setFromJDOM( org.jdom.Document jdom )
71 {
72     org.jdom.Element root = jdom.getRootElement();
73     String ID = (String)root.getAttribute("ID").getValue();
74     MCRObjectID mcr_id = new MCRObjectID(ID);
75     ci = new MCRClassificationItem(mcr_id.getId());
76     List element_list = root.getChildren();
77     org.jdom.Element categories = null;
78     int len = element_list.size();
79     for (int i=0;i<len;i++) {
80         org.jdom.Element tag = (org.jdom.Element)element_list.get(i);
81         if (tag.getName().equals("label")) {
82             String text = tag.getAttributeValue("text");
83             String desc = tag.getAttributeValue("description");
84             String lang = tag.getAttributeValue("lang",
85                 org.jdom.Namespace.XML_NAMESPACE);
86             ci.addData(lang,text,desc);
87         }
88         else {
89             categories = (org.jdom.Element)tag.clone(); }
90     }
91     LOGGER.debug(ci.toString());
92     LOGGER.debug("Element name = "+categories.getName());
93     cat = new ArrayList();
94     MCRClassificationObject [] pid = new
95     MCRClassificationObject[MAX_CATEGORY_DEEP];
96     int [] pos = new int[MAX_CATEGORY_DEEP];
97     List [] list = new List[MAX_CATEGORY_DEEP];
98     pid[0] = ci;
99     pos[0] = 0;
100    list[0] = categories.getChildren();
101    int deep = 0;
102    while(deep != -1) {
103        if (pos[deep] >= list[deep].size()) {
104            deep--; if(deep < 0) {break;}
105            pos[deep]++; continue; }
106        org.jdom.Element cattag = (org.jdom.Element)list[deep].get(pos[deep]);
107        String name = cattag.getName();
108        if (name.equals("label")) { pos[deep]++; continue; }
109        if (!name.equals("category")) { pos[deep]++; continue; }
110        String theID = cattag.getAttribute("ID").getValue();
111        MCRCategoryItem ci = new MCRCategoryItem(theID, pid[deep]);
112        List catlist = cattag.getChildren();
113        int catlen = catlist.size();
114        boolean catflag = false;
115        for (int i=0;i<catlen;i++) {
116            org.jdom.Element tag = (org.jdom.Element)catlist.get(i);
117            if (tag.getName().equals("label")) {
118                String text = tag.getAttributeValue("text");
119                String desc = tag.getAttributeValue("description");
120                String lang = tag.getAttributeValue("lang",
121                    org.jdom.Namespace.XML_NAMESPACE);
122                ci.addData(lang,text,desc);
123            }
124            else {
125                if (tag.getName().equals("url")) {
126                    String url = tag.getAttributeValue("href",
127                        org.jdom.Namespace.getNamespace("xlink", MCRDefaults.XLINK_URL));
128                    ci.setURL(url);
129                }
130                else {
131                    catflag = true; }
132            }
133        }
134        cat.add(ci);
135        if (catflag) {
136            pos[deep+1] = 0;
137            pid[deep+1] = ci;
138            list[deep+1] = catlist;
139            deep++;
140        }
141        else {
142            pos[deep]++; }
143    }
144 }

```

3.6.4 Fazit

In diesem Kapitel wurde deutlich, dass das prinzipielle Ziel der MyCoRe-Persistenz, Daten unabhängig von den Backends zu verwalten, erreicht wurde. So wurde mit keinem Satz auf Implementierungen der Backend-Schnittstellen eingegangen. Dies ist auch nicht Ziel der Untersuchung gewesen, weil diese Teile prinzipiell ohne Änderungen der MyCoRe-Anwendung optimierbar und austauschbar sind. Allerdings wurde auch deutlich, dass MyCoRe eher evolutiv gewachsen ist, als dass ein Gesamtkonzept implementiert wurde. Das zeigt sich an einigen Stellen, die widersprüchlich implementiert wurden. So geht aus der Konzeption eine $n : m$ -Beziehung zwischen Derivaten und Objekten hervor, also die prinzipielle Möglichkeit, ein Derivat mehreren Objekten zuzuordnen. Es sei dem Fakt geschuldet, dass diese Funktionalität offenbar nicht benutzt wurde, dass durch das Löschen eines Dokuments und damit seiner Derivate es zu keinen Fehlern in laufenden Systemen bislang gekommen ist. Es zeigt jedoch auch, dass eine klare Schnittstellenbeschreibung zwischen den Softwareschichten im Rahmen einer Gesamtarchitektur und die Einhaltung des Architekturkonzeptes (Abb. 3.1) solche Fehler verhindern helfen können. Im Verlauf dieses Kapitels wurde deutlich, welchen Zweck diese Komponentenaufteilung verfolgt und dass sie nicht vollständig umgesetzt worden ist.

In diesem Abschnitt wurde das Architekturbild aus Abbildung 3.1 „aufgeweicht“. Es wurde deutlich, dass das Ziel nicht erreicht wurde, diese Architektur im Code zu repräsentieren. Wie in Abschnitt 3.2 erläutert, ist die Klasse `MCRObject` das JAVA-Gegenstück zum XML-Code mit dem Wurzelement `<mycoreobject>`. Laut der Architektur müsste diese Klasse zur Schicht der Datenmodellklassen gehören. Sie stellt dennoch Methoden zur Verfügung, die ihre Persistenz betreffen. Es wurde gezeigt, wieso und wann dies ungünstig in Erscheinung tritt. Das Löschen stellt sich geradezu als Paradebeispiel für die negativen Folgen der Aufweichung der Softwarearchitektur aus Abschnitt 3.1 dar. Es wurde versäumt, eine Persistenzschnittstelle zu definieren, deren Implementierung das „Waisenkinderverbot“ befolgen muss. Das Ablaufdiagramm (Abb. 3.10) zeigt den Aufwand, der jetzt in der dafür nicht vorgesehenen Datenmodell-Schicht betrieben werden muss.

Architekturumsetzung

Abbildung 3.16 fasst das Ergebnis der Untersuchung des Persistenz-Systems in einem neuen Architektordiagramm zusammen, was gegenüber Abbildung 3.1 eher den tatsächlich ermittelten Gegebenheiten entspricht. Es verdeutlicht, dass Datenmodell und Persistenz keineswegs getrennt sind. Die XML-Präsentation ist fest in der Geschäftslogik verdrahtet. Auch der Zugriff auf die Persistenzsysteme erfolgt nicht über eine einheitliche Schnittstelle. Die „Persistenz-Schicht“ wird in Object-Store, Content-

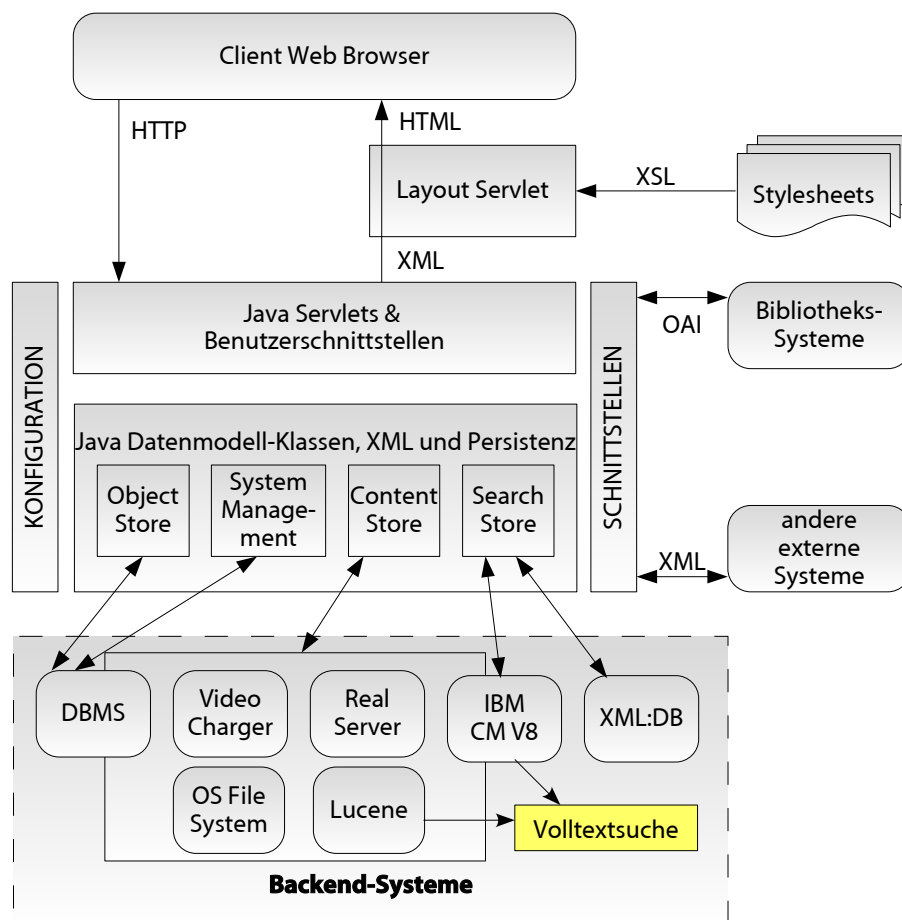


Abbildung 3.16: Architektur des MyCoRe-Systems (Realisierung)

Object-Store

Search-Store

Content-Store

Store und Search-Store unterteilt. Der Object-Store ist vornehmlich für die Aufnahme von Metadaten-Objekten gedacht; seine Schnittstelle ist das `MCRXMLTableInterface` (vgl. Abbildung 3.8). Um Suchanfragen (siehe Abschnitt 3.5) zu beantworten können, wird der Search-Store mit den Metadaten gefüttert, die auch schon der Object-Store besitzt; diesmal heißt die Schnittstelle `MCRObjectSearchStoreInterface` (vgl. Abbildung 3.9). Dateien landen davon unberührt in einem Content-Store (in Grafik extra gerahmt), der in 3.6.2 ausführlich beschrieben wurde. Für die Verwaltung all dieser „Stores“ wird ein relationales Datenbank-Management-System, wie IBM DB2 oder MySQL, benötigt. Es dient unter anderem als Verzeichnis für alle abgespeicherten Dateien und für Verknüpfung von Dokumenten untereinander, der Nutzerverwaltung und ist Basis der einzigen `MCRXMLTableInterface`-Implementierung.

Ein weiterer Aspekt der Architekturumsetzung ist die Schwierigkeit, mit der Funktionen eines generischen Repositories in MyCoRe integriert werden können. So wurden beim Design der Architekturkomponenten wichtige Funktionen wie Zugriffsregelungen, Versionierung und Transaktionen, nicht berücksichtigt. Das nachträgliche Einbauen wäre bei einer klaren, nachvollziehbaren Trennung zudem leichter möglich. Allerdings ermöglicht die freie Zugriffsmöglichkeit auf die Quellen von MyCoRe im Prinzip erst die Erweiterung für jedermann. Mit kleineren Abstrichen im Funktionsumfang ist es also schon heute möglich, herstellerunabhängig, Anwendungen für Dokumentenserver zu erstellen, wie eine Vielzahl von Projekten¹⁴ zeigt.

¹⁴<http://www.mycore.de/content/main/information/projects.xml>

Kapitel 4

Content Repository Schnittstelle: JSR-170

Die dieser Diplomarbeit zugrunde liegende Aufgabenstellung sieht nach der Analyse der MyCoRe-Persistenz-Schicht den Vergleich mit JSR-170 (Nüscher u. a. 2005) vor. In diesem Kapitel, in dem der JAVA-Content-Repository-Standard vorgestellt wird, wird geklärt, was JSR-170 ist und bedeutet.

4.1 Einführung

JSR steht für „*Java Specification Requests*“, also einen Sachverhalt für die JAVA-Welt zu klären. Es ist ein offener Prozess, der JAVA Community Process¹, in dem solche Spezifikationen erarbeitet werden. JSR ist am ehesten vergleichbar mit den berühmten RFC², die Standards in der Welt des Internets regeln.

JSR-170 ist also die 170. Aufforderung, jedoch nicht ausschließlich zu diesem Thema, für JAVA einen Standard zu schaffen. Als Ziel steht eine *Content Repository API for JAVA Technology* (JCR), also die Möglichkeit, über eine festgelegte Schnittstelle auf Content Repositories zugreifen zu können. Das World Wide Web ist das beste Beispiel dafür, dass einfache standardisierte Schnittstellen erfolgreich sind (Fielding 2005). Es gibt bereits Standards in diesem Bereich, populäre Vertreter sind z.B. ICE (Brodsky u. a. 1999) und WebDAV (Goland u. a. 1999), diese sind jedoch auf Protokoll-Ebene festgelegt.

¹<http://www.jcp.org/>

²<http://www.rfc-editor.org/rfc.html>

„Content Repositories are key elements in the software infrastructure of global companies. Infrastructure however can only be successful when it is based on standards. So far this has been missing in the industry which has been characterized by a confusing plethora of proprietary repository technologies. The goal of JSR 170 is to change this. “

(Day Software 2004, David Nüscheler)

Traditionell ist die Content-Management-Branche breitgefächert, so gibt es allein auf dem deutschen Markt knapp 300 Produkte³, davon knapp 100 im Bereich *Enterprise Content Management*.

In einer Untersuchung (Gartner, Inc. 2004) wurde festgestellt, dass bereits eine Vielzahl von Content-Management-Systemen in Betrieb sind, die ursprünglich für einzelne Abteilungen angeschafft wurden. Deren Einsatzzweck muss jetzt vielfältig erweitert werden. Aber die momentane Situation ist, dass viele Firmen eine ganze Reihe von datenbunkernden Systemen im Einsatz haben, zu denen sie jetzt Zugriff brauchen. Doch ihre Daten sind förmlich „luftdicht“ in verschiedenen Content-Management-Systemen versiegelt, welche ihre Geschäftsgrundlage zur Anschaffungszeit gehabt hatten. Einige Anbieter stellen zwar eine API bereit, um den Inhalt für externe Systeme anzubieten. Dieses sind jedoch immer proprietäre Ansätze. Andere beziehen die Daten für ihre Portale von verschiedenen CM-Systemen unterschiedlicher Hersteller. Grundsätzlich gibt es jedoch keine verfügbaren hersteller- und plattformunabhängigen Lösungen. Es ist ersichtlich, dass ein offenes System größere Flexibilität und Stabilität besitzt und immun gegenüber Entwicklungsstillständen ist, sollte der Hersteller das Geschäft aufgeben. Es hat den Vorteil über längere Zeit größere Stabilität und Interoperabilität zu sichern, wenn der offene Standard eine breite Unterstützung findet. Bezogen auf JSR-170 sieht man vor allem die Aufspaltung in zwei Level der Konformität positiv. So ist eine Implementierung der Basisfunktionen selbst für kleinere Hersteller problemlos zu bewältigen. JSR-170 löst jedoch nicht die Problemfelder *Content-Integration*, *Business Process Management* und *Application-Integration*, aber es stellt die Technologie bereit, die von Systemen in diesen Bereichen genutzt werden können.

Notwendigkeit

³<http://www.contentmanager.de/itguide/marktuebersicht.html>

Im folgenden werden JSR-170 und JAVA Content Repository API (kurz: JCR) synonym verwendet, da beide Namen desselben Standards sind. Es gibt unterschiedliche Ansätze Schnittstellen bereitzustellen: kontrollorientiert und inhaltsorientiert. Der kontrollorientierte Ansatz bietet dem Programmierer eine sehr genaue Kontrolle über die Manipulation der Daten und kann spezielle Funktionen eines Herstellers unterstützen und so effektiver sein. Er kontrolliert das WIE der Manipulation. Währenddessen der Programmierer des inhaltsorientierten Ansatzes darauf vertrauen muss, dass der Hersteller diese Schnittstelle effektiv implementiert. Hier steht das WAS im Vordergrund. Letzteres muss aber kein Nachteil sein: Erst dieser Ansatz erlaubt es doch erst, dass eine breite Basis an zusätzlichen Tools geschaffen wird, die womöglich zusammen mehr Funktionalität bringen, als es ein einzelner Hersteller bieten kann. Richtig zum Tragen kommen die Vorteile, wenn mehrere Applikationen zu einer Anwendung kombiniert werden. So ist der Aufwand bei der kontrollorientierten Lösung $O(n^2)$, während er bei der inhaltsorientierten Lösung weiterhin bei $O(n)$ liegen würde. Ziel von JSR-170 ist es, den Zugriff auf Content-Repositories zu standardisieren. *Herangehensweise*

Content-Repository

Ein Content Repository ist ein generischer Daten „Super-Speicher“ für Applikationen, sowohl für kleine, wie auch für große Daten. Es ermöglicht die Manipulation und Speicherung von strukturierten und unstrukturierten Inhalten, binären und Textformaten, Metadaten und dynamischen Beziehungen. Wünschenswert sind zudem Dienste wie einheitliche Zugriffskontrolle, Sperren, Transaktionen, Versionierung, Überwachung und Suche.

(Fielding 2005, Seite 4)

4.2 Repository-Modell

Kernstück der JCR-Spezifikation ist das Repository-Modell. Ein Repository wird als allgemeiner Datenspeicher angesehen, der unabhängig von der Art der zu speichernden Daten verwendet werden soll (Fielding 2005, Seite 5). Der Entwickler einer Anwendung soll sich nicht mehr darum kümmern, wie die Daten gespeichert werden, sondern er benutzt für alle anfallenden, z.B. Konfigurationseinstellungen, Daten und Metadaten, eine Schnittstelle. Im Nachhinein, also zur Einsatzzeit der Software und nicht zur Programmierzeit, entscheidet er sich dann für spezialisierte Repositories, die einzelne Daten besonders effizient verarbeiten können.

Ein Repository besteht aus einer theoretisch unbegrenzten Zahl von Arbeitsbereichen (engl. *Workspace*), die ihrerseits eine hierarchische Struktur von

Elementen (*item*) in Form von Knoten (*node*) und Eigenschaften (*property*) besitzen. Elemente sind eine Generalisierung von Knoten und Eigenschaften. Im folgenden wird daher im Zusammenhang mit JCR von Elementen gesprochen, wenn dies sowohl für Knoten als auch Eigenschaften zutreffend ist. Über die Knoten bekommt der Arbeitsbereich seine hierarchische Darstellung, während die Eigenschaften die Informationsträger sind. Eigenschaften sind atomar, das heißt eine Eigenschaft ist ein Name-Wert-Paar. Während Knoten noch Kinderelemente enthalten können, ist dies nicht für Eigenschaften möglich. Diesen Zusammenhang verdeutlicht noch einmal **Abbildung 4.1**. Da Waisenkinder nicht erlaubt und Eigenschaften immer Kinder von Knoten sind, ergibt sich, dass das Wurzelement des Arbeitsbereichs ein Knoten sein muss.

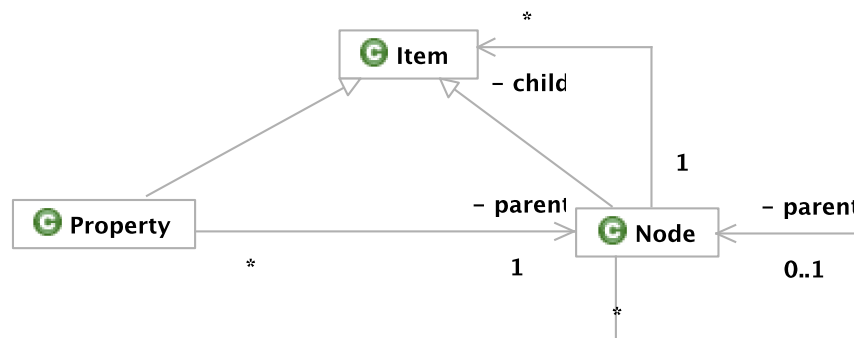
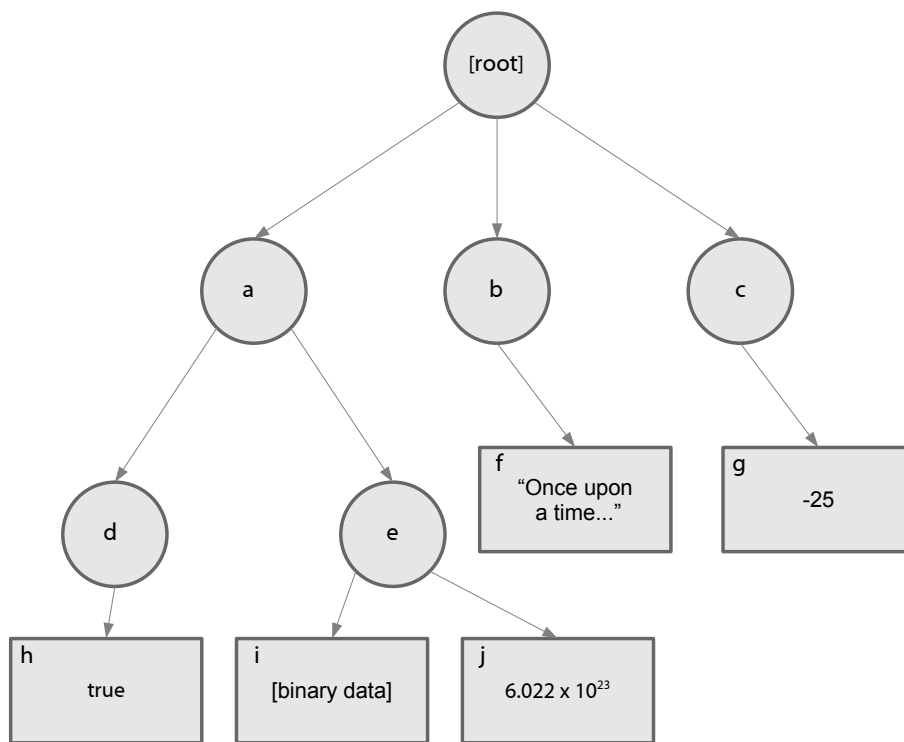


Abbildung 4.1: Klassendiagramm: Item, Node und Property

Abbildung 4.2 zeigt eine Elementhierarchie eines beliebigen Arbeitsbereichs. Die Knoten ($[root], a, \dots, e$) bauen eine Hierarchie auf, bei der es einen dedizierten Knoten gibt: den Wurzelknoten. Dieser besitzt als Kinder die Knoten **a**, **b** und **c**. Ersterer besitzt noch zwei weitere Knoten als Kinder: **d** und **e**. Knoten **e** wiederum hat zwei Eigenschaften. Eigenschaft **i** enthält als Wert beliebige binäre Daten, z.B. ein Bild, während die Eigenschaft mit dem Namen **j** eine Fließkommazahl enthält: $6,022 \cdot 10^{23}$. Es ist nicht wichtig wie diese Ansicht letztendlich gespeichert wird, vielmehr geht sie nur aus der JCR-Spezifikation hervor, die Herstellern bezüglich der Implementierung viele Spielräume einräumt.

Über diese Freiräume will man erreichen, dass eine möglichst breite Unterstützung des Standards erreicht wird. Um eine Implementierung zu erleichtern bietet man unterschiedliche Stufen der Konformität an (*Compliance Level*). So



Quelle: (Nüscheler u. a. 2005)

Abbildung 4.2: Hierarchie: Arbeitsbereich

verlangt man von einer Implementierung der Stufe 1, dass auf das Repository lesend zugegriffen werden kann. Dieser Schritt erlaubt bereits den Export (via XML) aller Daten des Repositories in eines mit Schreibzugriff (Level 2). Die folgenden Abschnitte stellen die erforderlichen Funktionen in den jeweiligen Stufen der Konformität vor, **Abbildung 4.3** zeigt die Klassen und Methoden, die darin besprochen werden.

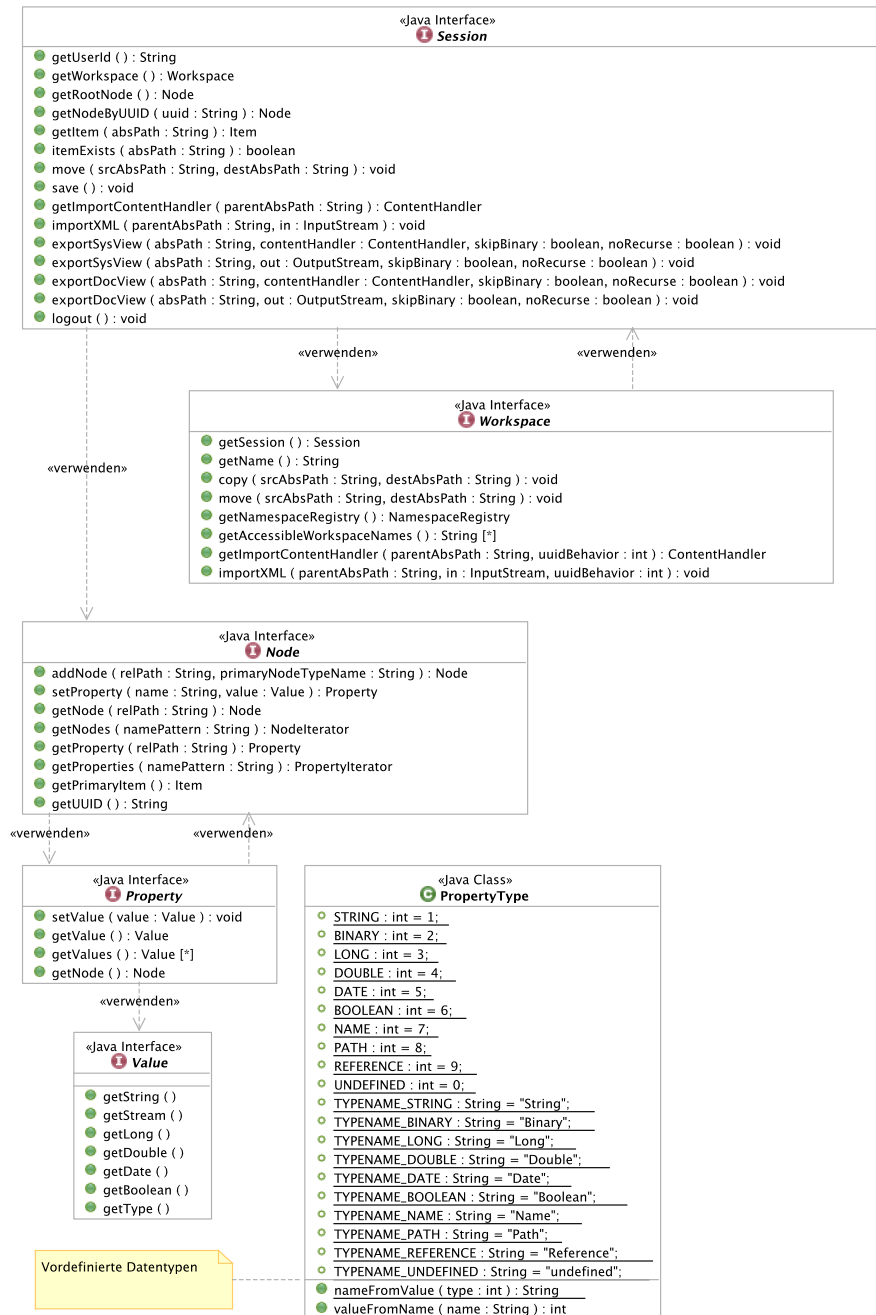


Abbildung 4.3: Klassendiagramm: Session, Workspace, Elemente und Werte

4.2.1 Level 1 Funktionen

Wie erwähnt, enthält der Level 1 nur Funktionen zum lesenden Zugriff. Wozu soll man ausschließlich lesend auf ein Repository zugreifen? Neben der Exportmöglichkeit reicht diese Stufe auch aus, um Portalsoftware zu betreiben, die ihre Informationen aus verschiedenen Quellen beziehen, aber nicht ändern. Bieten alle Repositories des Portals eine JCR-Schnittstelle an, wird der Entwicklungsaufwand für die Portalsoftware reduziert, da nur diese Schnittstelle unterstützt werden muss. So gesehen ist also eine Level-1-Implementierung sehr wohl relevant, weil man schon mit dieser Möglichkeit externe Systeme anschließen kann.

Level 1 beinhaltet folgende Funktionen (Nüscheler u. a. 2005, Kapitel 4.2,6):

- **Sitzung innerhalb eines Arbeitsbereichs initiieren.** Um mit dem Repository zu arbeiten, muss sich der Client gegenüber dem Repository vorstellen. Er übergibt seine Autorisierungsdaten (*Credentials*) und sucht sich seinen Arbeitsbereich (*Workspace*) aus, um sich anzumelden. Letzterer ist über seinen eindeutigen Namen dem Repository bekannt. Eine Liste aller verfügbaren Arbeitsbereiche hält das Repository zur Auswahl bereit (`Workspace.getAccessibleWorkspaceNames`). Da für die gesamte Sitzung alle Aktionen einem Nutzer (über die Sitzung) zugeordnet werden können, laufen diese auch im Kontext dieses Nutzers. Es ist möglich, dass der Nutzer von vornherein nur die Daten sieht, auf die er überhaupt zugreifen kann. Dieses Verhalten ist aber implementierungsabhängig und nicht Gegenstand von Level 1.
- **Durchqueren des Baums.** Sobald die Sitzung initiiert wurde, besteht die Möglichkeit, über den Wurzelknoten den gesamten Baum zu durchqueren, sich also jeweils von einem Knoten seine Kinder geben zu lassen, solange bis man schließlich auf Blattebene angekommen ist. Ferner ist es möglich, sowohl Knoten als auch Eigenschaften direkt mittels Pfadangaben zu erreichen, z.B. „/a/e/j“ liefert bei Abbildung 4.2 die Eigenschaft mit dem Wert $6,022 * 10^{23}$. Neben diesem Pfadzugriff erlauben es bestimmte Knoten – wenn diese als referenzierbar gekennzeichnet sind – über einen eindeutigen Bezeichner, der UUID⁴, direkt angefordert zu werden. Jede dieser Zugriffsmöglichkeiten hat seine Vor- und Nachteile, jedoch sind diese Funktionalitäten je nach Anwendungszweck und Operation oft notwendig. Über die Nutzung entscheidet der Client – und damit der Anwendungsprogrammierer.

⁴Universal Unique Identifier

- **Knoten und Eigenschaften auslesen.** Beim Zugriff auf die Elemente des Baums kann man diese auslesen. Da der gesamte Inhalt des Repositories in den Eigenschaften steckt, liefern Knoten nur benannte Strukturinformationen. Jeder Eigenschaft ist dabei ein Typ zugewiesen. Es gibt eine Zahl vorgegebener Typen, aus denen einer gewählt werden kann. Diese Liste umfasst die Typen String (`PropertyType.STRING`), binäre Daten (`PropertyType.BINARY`), Datum (`PropertyType.DATE`), Ganzzahlen (`PropertyType.LONG`), Fließkommazahlen (`PropertyType.DOUBLE`), Boolesche Werte (`PropertyType.BOOLEAN`), Name (`PropertyType.NAME`), Pfad (`PropertyType.PATH`) und Referenz (`PropertyType.REFERENCE`). Die letzten drei liefern Möglichkeiten jenseits der Vater-Sohn-Beziehung zusätzliche Zeiger zu anderen Elementen zu setzen. Dabei bietet nur der letzte Typ die Möglichkeit die referentielle Integrität sicherzustellen. Er erfordert aber, dass der Ziel-Knoten als referenzierbar gekennzeichnet ist, da eine Eigenschaft vom Typ Referenz eine UUID enthält, die nur solche Knoten bereitstellen. Da JCR-Implementierungen nur zwischen diesen Typen differenzieren, gibt es keine weitere Unterscheidung zwischen Daten und Metadaten, dies ist Aufgabe des Anwendungsentwicklers. Dieser hat jedoch die Möglichkeit, einen speziellen Kindknoten als Primärkind zu deklarieren, wenn der Vaterknoten dies zulässt. Dies kann man dafür nutzen, um zum Beispiel Metadaten grundsätzlich unter den Primärknoten abzulegen, um so Metadaten von Nutzdaten auch strukturell zu trennen.
- **XML-Export.** JCR unterstützt das Abbilden des Repository-Modells in XML. Dabei werden zwei unterschiedliche Abbildungsmöglichkeiten angeboten. Zum einen eine Systemsicht, die alle Informationen enthaltend einen kompletten Arbeitsbereich in XML übersetzt. Da diese Sicht für den Menschen schwer verständlich sein kann, existiert zudem eine Dokumentsicht, die aber bessere Lesbarkeit durch Informationsverlust erkaufte. Die Dokumentsicht ist das Format, das zudem für XPATH-Anfragen verwendet wird.
- **Abfragemöglichkeit mit XPATH-Syntax.** Als Abfragesprache für XML ist XPATH längst verbreitet (Clark u. DeRose 1999). JCR bietet deshalb diese akzeptierte Syntax als Abfragemöglichkeit an, da es eine Abbildung zwischen Repository-Modell und XML gibt. Eine XPATH-Suche wird dabei auf der Dokumentsicht ausgeführt und liefert entweder eine Tabelle oder Liste mit Eigenschaften als Ergebnisse der Anfrage.

- **Vorübergehende Namensraum-Neuzuordnung.** Um Namenskollisionen zu vermeiden, wurde für XML das Konzept der Namensräume (*Namespaces*) definiert (Bray u. a. 1999). Dabei können Namen ein Präfix enthalten, das durch einen Doppelpunkt abgetrennt ist. Dieses Präfix ist einer URI⁵ zugeordnet, dem Namensraum. JCR adaptiert dieses Konzept für Knoten und Eigenschaften. Jedes konforme Repository stellt dafür eine *Namespace-Registry* bereit, das jedem Präfix auf eine URI abbildet. In Level-1-Repositories kann man diese Abbildung zeitweise, innerhalb einer bestimmten Sitzung, überschreiben. Dies gilt jedoch nicht für vordefinierte Präfixe, die für JCR reserviert sind. **Tabelle 4.1** stellt diese kurz vor.

Namensraum	Beschreibung
jcr	reserviert für Elemente, die von vordefinierten Knotentypen definiert wurden
nt	reserviert für vordefinierte (primäre) Knotentypen
mix	reserviert für vordefinierte „mixin“ Knotentypen
sv	reserviert für das Abbilden eines Arbeitsbereichs in XML
„“	leeres Präfix; ist der Standard-Namensraum

Tabelle 4.1: Vordefinierte Namenräume in JSR-170

- **Feststellen vorhandener Knotentypen.** Jeder Knoten in JCR muss genau einen Primärtyp besitzen, dieser regelt Einschränkungen bezüglich der Kinderelemente, z.B. welche Eigenschaften der Knoten haben darf oder muss. Neben dem Primärtyp darf ein Knoten aber noch einen oder mehrere so genannte „mixin“-Typen haben, welche dem Knoten zusätzliche Merkmale zuweisen, ähnlich der des Primärtyps. Level 1 erlaubt es, sowohl Primärtyp und „mixin“-Typen festzustellen (Eigenschaften: **jcr:primaryType**, **jcr:mixinTypes**), als auch zu entdecken, welche Typen vom Repository unterstützt werden und wie diese definiert sind.
- **Feststellen von Zugriffsbeschränkungen.** Ein JCR-Repository kann Zugriffskontrollen unterstützten (siehe **Abbildung 4.4**). Neben der Möglichkeit dem Nutzer (bekannt durch seine *Credentials*), eine eigene Sicht auf den Arbeitsbereich zu geben und so von vornherein Elemente zu verbergen, die er nicht lesen darf, besteht ferner die Möglichkeit bestimmte Aktionen auf Rechte des Nutzers von der Anwendung zu überprüfen. Da das Login gegenüber dem Repository auch ohne *Credentials* gesche-

⁵Uniform Resource Identifiers

hen kann, können externe Authentisierungs- und Autorisierungsdienste z.B. über JAAS⁶ angebunden werden.

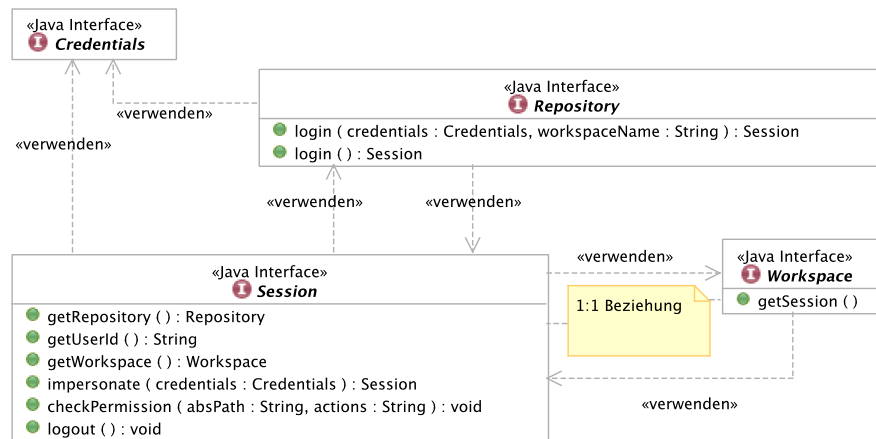


Abbildung 4.4: Klassendiagramm: Zugriffskontrolle in JCR

Das Ziel bei der Definition der Level-1-Funktionen war es, Anwendungen, die nur einen Lesezugriff benötigen, eine einfache, standardisierte Schnittstelle zum Datenzugriff zur Verfügung zu stellen. Während echte Content Management Systeme nicht ohne Level-2-Unterstützung auskommen werden, kann man die Implementation von Level 1 als einfachen, im Sinne von einem einfach zu implementierenden Schritt sehen, hin zu einem alle Funktionen umfassenden Repository.

4.2.2 Level 2 Funktionen

Level 2 bietet die Möglichkeit, dauerhafte Änderungen am Repository vorzunehmen. Neben dem Import von fremden Datenquellen kommen noch andere nützliche Funktionen hinzu, die im folgenden beschrieben werden (Nüscheler u. a. 2005, Kapitel 4.2,6.4.1,7).

- **Hinzufügen und Entfernen von Knoten und Eigenschaften.** Level 2 führt Methoden ein, um in dem Repository Knoten und Eigenschaften hinzuzufügen, zu entfernen, zu verschieben oder zu kopieren. Neben diesen Basisoperationen gibt es fortgeschrittene Methoden für das Verschieben, Kopieren und Klonen von Elementen zwischen verschiedenen Arbeitsbereichen. Es können sowohl einzelne Änderungen (pro Element), als auch

⁶<http://java.sun.com/products/jaas/>

alle (in einem Arbeitsbereich) gespeichert oder verworfen werden. Der JCR-Client kann zudem den Unterschied zwischen den aktuell ungespeicherten Änderungen und den Elementen im Workspace feststellen, um z.B. die Änderungen eines anderen Clients zu entdecken und diese mit seiner lokalen Kopie abzugleichen, zu überschreiben oder zu übernehmen.

- **Schreiben der Werte von Eigenschaften.** Neben den Strukturinformationen des letzten Punkts liegen im Repository vor allem die zu verwaltenden Inhalte in Form von Eigenschaftswerten. Diese Werte können im Level 2 geändert werden, dabei übernimmt das Repository automatisch die Konvertierung zwischen den verschiedenen Eigenschaftstypen.
- **Dauerhafte Änderung von Namensräumen.** Neben den Änderungen, die bereits in Level 1 möglich sind, erlaubt Level 2 Namensräume hinzuzufügen, zu entfernen und zu ändern. Diese Änderungen sind persistent in der Namespace-Registry möglich. Ausgenommen sind die eingebauten Namensräume, die für JCR erforderlich sind (siehe dazu Tabelle 4.1).
- **Import von XML.** Level-2-Repositories müssen den Import von XML in Systemsicht und Dokumentsicht erlauben. Wenn Daten in XML-Systemsicht vorliegen, gleicht dieser Vorgang einem Restore-Vorgang, weil es zwischen Repository und XML in Systemsicht eine eindeutige Abbildung gibt, andernfalls wird das XML-Dokument behandelt, als sei es in Dokumentsicht. Dafür werden die Namensräume automatisch in die Namespace-Registry übernommen und ein Baum von JCR-Knoten aufgebaut, der den Baum der XML-Elemente repräsentiert. XML-Attribute werden zu JCR-Eigenschaften. Zusätzlich werden eventuell noch nötige JCR-Eigenschaften erstellt (z.B.: `jcr:primaryType`, etc.). Die Namen für Elemente und Attribute aus dem XML-Dokument werden 1:1 in das Repository übernommen. Weitere Details und ein Beispiel zum Import und Export folgen in Abschnitt 4.4.
- **Zuweisen von Knotentypen zu Knoten.** Konforme Repositories müssen das Zuweisen von primären und „mixin“ Knotentypen bei der Erstellung von Knoten zulassen. Optional ist das Zuweisen und Entfernen von „mixin“ Knotentypen bei bereits existierenden Knoten. Die Zuweisung von Knotentypen kann auch über die Knotendefinition des Vaters automatisch erfolgen. Wenn die Definition des Vaters z.B. vorzieht, dass alle Kinderknoten vom Typ „myApp:docPage“ sein müssen, dann wird dieser Typ neuen Kindern automatisch zugewiesen, er muss nicht explizit angegeben werden. Anderslautende Zuweisungen müssen eine Ausnah-

me werfen. Knotentypdefinitionen sind daher ähnlich einem Schema in XML. In Abschnitt 4.3 werden Knotentypen weiter vorgestellt.

Ziel von Level 2 war es, schreibenden Zugriff auf ein Repository zu ermöglichen. In dem man gesonderte Funktionen als optional ausgelagert hat, konnte man so den Aufwand für eine Implementierung von Level 2 verringern. Darunter fallen dann besonders jene, die ein modernes Content Management System ausmachen.

4.2.3 Optionale Funktionen

Neben den Funktionen von Level 1 und 2 stellt JCR noch Standardschnittstellen für zusätzliche Funktionen bereit. Dazu gehören: Transaktionen, Versionierung, Überwachung und das Sperren von Objekten (Nüscheler u. a. 2005, Kapitel 4.2,8). Keine dieser Funktionen ist jedoch abhängig von einem bestimmten Implementierungsstand des Repositories. Das bedeutet, auch ein Level 1 Repository kann Versionierung unterstützen – lesend selbstverständlich.

- **Transaktionen** können von einem JCR-konformen Repository unterstützt werden. Die Implementierung muss sich an den JAVA-Standard JTA halten (Cheung u. Matena 2002). Dort wird erfasst, wie die Transaktionsgrenzen definiert werden, jedoch nicht in der JCR-Spezifikation. JTA kennt zwei Ansätze von Transaktionen: *container-managed* und *user-managed*. Im ersten Fall wird die Transaktionsverwaltung vom Anwendungs-Server wahrgenommen. Dies geschieht völlig transparent für den Client. Bei dem zweiten Fall, den *user-managed* Transaktionen, kann der Client die Transaktionsgrenzen innerhalb der Anwendung setzen. Falls eine JCR-Implementierung Transaktionen unterstützt, muss sie beide Ansätze anbieten.
- **Versionierung** erlaubt es, den Status eines Knotens festzuhalten, zu sichern und wiederherzustellen. Das Versionierungssystem von JCR ist nach dem *Workspace Versioning and Configuration Management* Standard modelliert, wie er in JSR-147 definiert ist (Clemm u. a. 2003). Ein konformes Repository besitzt, neben einer Zahl von Arbeitsbereichen, einen speziellen „*version storage*“-Bereich. Dieser besteht aus „*version histories*“. Genau dann, wenn zwei Knoten aus unterschiedlichen Arbeitsbereichen die gleiche UUID besitzen, sie also korrespondierende Knoten sind, teilen sie sich dieselbe *version history*. Eine Version-History ist eine Sammlung von Versionen, die über eine Nachfolger-Beziehung miteinander verbunden sind. Neue Versionen eines Knoten werden in die Version-History eingefügt, wenn der Knoten eingchecked wird (*check-in*). Versionierbare Knoten sind

grundsätzlich schreibgeschützt. Um mit der Hilfe von Level-2-Funktionen Änderungen vorzunehmen, muss er Knoten zunächst ausgecheckt werden (*check-out*), bei dem der Schreibschutz über den Knoten und über den nicht-versionierbaren Teil seines Unterbaums aufgehoben wird, siehe auch Abschnitt 4.3.

- **Überwachung von Knoten – Ereigniskontrolle.** Diese Funktion erlaubt es Applikationen Veränderungen am Arbeitsbereich wahrzunehmen. Dazu registrieren sie sich für interessante Ereignisse, z.B. das Hinzufügen eines neuen Knotens, um auf diese reagieren zu können. Diese Ereignisse werden nur ausgelöst, wenn Änderungen persistent am Arbeitsbereich vorgenommen werden, jedoch nicht, wenn sie in einer Sitzung anhängig sind.
- **Sperren** erlauben es Nutzern, vorübergehend Knoten zu sperren, um anderen Nutzern das Ändern der Knoten zu verweigern. Diese Funktion erlaubt das Serialisieren der Zugriffe zu einem Knoten in einem Multi-User-System und verhindert unter anderem so das bekannte „*Lost Update*“-Problem. JCR verhindert auch ohne Locking das versehentliche Überschreiben von Repository-Inhalten durch eine **InvalidItemStateException**, jedoch verhindert *Locking*, dass diese Ausnahme auftritt. Um festzustellen, ob *Locking* unterstützt wird, reicht es zu überprüfen, ob der Knotentyp „**mix:lockable**“ vom Repository zur Verfügung steht.
- **Abfragemöglichkeit mit SQL Syntax.** Als Abfragesprache ist SQL vor allem im relationalen Datenbankumfeld weit verbreitet. Da es möglich ist, dass ein Repository auf einer solchen Datenbank aufgesetzt wird und um Anwendern einen schnellen Einstieg zu ermöglichen, kann ein Repository SQL als Abfragesprache unterstützen. Dafür wird das Repository, ähnlich wie für XPATH in Dokumentsicht, in eine Datenbanksicht überführt.

Optionale Funktionen erlauben es einerseits das breite Spektrum an Repository-Funktionalitäten abzudecken und diese durch eine standardisierte Schnittstelle anzusprechen. Jedoch können andererseits Implementierungshürden kleingehalten werden. So passt sich dieses in das Gesamtkonzept der JCR-API und ihre Trennung in Level 1, Level 2 und optionale Funktionen ein.

4.3 Knotentypen

Ein wichtige Funktion in Repositories kann es sein, verschiedene Entitäten in ihrem Typ zu unterscheiden, zum Beispiel, damit ein Administrator Kontrolle

darüber erhalten kann, welchen Integritätsbedingungen Daten entsprechen müssen, die in ein Repository eingestellt werden. Über ihren Typ sind Einschränkungen möglich. Ähnliches wird bei XML über XML-Schemas und bei Datenbank-Management-Systemen über das Datenbankschema realisiert. In einem JCR-Repository gibt es dafür die Knotentypen (*node types*), die in diesem Abschnitt näher beschrieben werden.

4.3.1 Knotentypfunktionen in JCR-Levels

In Abschnitt 4.2 wurde bereits die funktionale Trennung der JCR-Repositories in unterschiedliche Levels der Konformität vorgestellt. Diese wird hier für den Bestandteil der Knotentypen verfeinert.

Level 1 spezifiziert die folgenden Abfragemöglichkeiten bezüglich der Knotentypen:

- **Primär- und Mixin-Knotentypen.** In Repositories des Level 1 ist es möglich, von vorhandenen Knoten den primären Knotentyp festzustellen oder die „mixin“ Knotentypen (vgl. Abschnitt 4.2.1).
- **Auflistung der Knotentypen.** Es ist einem Client möglich, sämtliche Knotentypen eines bestimmten Repositories in Erfahrung zu bringen.
- **Abfrage einer Knotentypdefinition.** Es ist einem Client möglich, die Definition von einem unterstützten Knotentyp zu lesen.
- **Feststellen von Einschränkungen.** Sollte über den Knotentyp des Vaters eines Knoten Einschränkungen auf diesen oder seine Eigenschaften existieren, so können diese festgestellt werden.

Level 2 spezifiziert zusätzlich folgende Möglichkeiten bezüglich der Knotentypen:

- **Zuweisen von Primärtyp.** Beim Erstellen von Knoten kann diesem ein Primärtyp zugewiesen werden.
- **Zuweisen von „mixin“ Knotentypen.** Es ist einem Client möglich zusätzliche „mixin“ Knotentypen zuzuweisen.

4.3.2 Definition von Knotentypen

Die Spezifikation (Nüscheler u. a. 2005, Kapitel 6.7) lässt die Frage explizit offen, wie neue Knotentypen definiert werden. Es ist den Implementierern überlassen, das Erstellen von eigenen Knotentypen zu ermöglichen. Dies wird damit begründet, dass die Zahl der Ansätze, Entitäten Typen zuzuweisen, so

vielfältig ist, dass es schwierig ist, dafür einen einzelnen Mechanismus zu definieren.

Die Ermittlung von Knotentypen, wie in Abschnitt 4.3.1 bereits skizziert, ist dagegen sehr umfangreich spezifiziert, so dass auf diese im folgenden weiter eingegangen wird. Um Information bezüglich eines Knotentyps über Methoden (siehe **Abbildung 4.5**) bereitzustellen, wurde eine Liste von diesen Informationen erstellt, die zugänglich sein müssen:

- **Name.** Jeder Knotentyp muss mit einem im Repository eindeutigen Namen definiert sein. Die Namenskonventionen orientieren sich an denen von Elementen (*items*). Das heißt, durch einen Doppelpunkt abgetrennt, können Namen zu einen Namensraum zugeordnet werden, um Namenskonflikte zu vermeiden (siehe Seite 57). Alle vordefinierten primären Knoten der JCR-Spezifikationen haben den Präfix **nt**. Vordefinierte „mixin“ Knotentypen besitzen das Präfix **mix**.
- **Supertypen.** Ein Knotentyp kann einen anderen Knotentyp erweitern.
- **Mixin Status.** Ein Knotentyp ist entweder ein „mixin“ Knotentyp oder ein primärer Knotentyp. Diese Information ist in der Knotentypdefinition verankert.
- **Anordbare Kinderknotenstatus.** Ein primärer Knotentyp kann definieren, dass die Reihenfolge der Kindknoten vom Client festgelegt wird. Wenn der Status `true` ist, dann ist die Unterstützung dafür (bereitzustellen mittels `Node.orderBefore()`) erforderlich, sonst (`false`) nur möglich. Nur bei einem Primärtyp ist dieser Status von belang.
- **Eigenschaftendefinition.** Ein Knotentyp enthält eine Menge von Definitionen, welche Eigenschaften, mit welchen Merkmalen, ein Knoten haben muss oder darf.
- **Kindknotendefinition.** Ein Knotentyp enthält eine Menge von Definitionen, welche Kindknoten, mit welchen Knotentyp, ein Knoten haben muss oder darf.
- **Name des primären Elements.** Ein Knotentyp kann spezifizieren, welches Kindelement, Knoten oder Eigenschaft, das primäre Element ist. Diese Hinweise wird von `Node.getPrimaryItem()` ausgewertet.

Mit diesen Daten ist in Level-1-Repositories, obwohl jegliche Schreibunterstützung fehlt, die Möglichkeit vorhanden, Informationen über den Schreibzugriff zu erhalten, z.B. wenn Kindknoten frei anordbar sind. Generell wird bei der Knotentypdefinition nicht zwischen Level 1 und 2 unterschieden, da beide

in jedem Fall nur lesend auf die Definitionen zugreifen. Es ist möglich, dass Knotentypen in Repositories mit unterschiedlichem Konformitätslevel unterstützt werden. Es ist gewünscht, bezüglich des Levels in der Knotendefinition nicht zu unterscheiden, auch wenn in Level-1-Repositories nur lesend auf das Repository zugegriffen werden kann.

Auf JAVA-Sicht bezogen, kann man primäre Knoten als Klassen verstehen, die instanziiert werden. Den Knoten entsprechen dann die Objekte. Das Äquivalent zu den „mixin“ Knotentypen, die nicht instanzierbar sind, sind die Schnittstellen (*interface*) in JAVA, die den Klassen zusätzliche Funktionen „zur Verfügung“ stellen.

Tabelle 4.2 zeigt beispielhaft die Definition des Knotentyps `nt:base`. Obwohl die Spezifikation noch andere primäre Knotentypen definiert, ist dies der einzige, den alle Repositories bereitstellen müssen. Jeder andere Knoten muss ein Subtyp von `nt:base` sein. Aus der Definition und der Notwendigkeit, dass `nt:base` Supertyp jeden anderen Primärtyps ist, geht hervor, wie jedem Knoten primäre und „mixin“ Knotentypen zugeordnet werden: über die Eigenschaften `jcr:primaryType` und `jcr:mixinTypes`. Deren Definition entsprechen den bereits besprochenen Eigenschaften.

nt:base		
Name	Wert	
NodeTypeName	nt:base	
Supertypes	[]	
IsMixin	false	
HasOrderableChildNodes	true	
PrimaryItemName	null	
PropertyDef	Name	jcr:primaryType
	RequiredType	NAME
	ValueConstraints	[]
	DefaultValue	null
	AutoCreate	true
	Mandatory	true
	OnParentVersion	COMPUTE
	Protected	true
	Multiple	false
PropertyDef	Name	jcr:mixinTypes

Fortsetzung auf nächster Seite

Tabelle 4.2 – Fortsetzung von voriger Seite

Name	Wert
RequiredType	NAME
ValueConstraints	[]
DefaultValue	null
AutoCreate	false
Mandatory	false
OnParentVersion	COMPUTE
Protected	true
Multiple	true

Tabelle 4.2: Knotentypdefinition: `nt:base`

Obwohl es explizit der Implementierung überlassen wird, wie Knotentypdefinitionen abgespeichert werden, empfiehlt der Standard dies auch im Repository zu tun. Zur Definition stehen die im Standard definierten Knotentypen `nt:nodeType`, `nt:propertyDefinition` und `nt:childNodeDefinition` zur Verfügung. Implementierungen, die diesem Vorschlag folgen, sollten alle Knotentypdefinitionen als Knoten unterhalb von `/jcr:system/jcr:nodeTypes` ablegen. Ein Export in Dokumentsicht von dem in Tabelle 4.2 vorgestellten Knotentyp zeigt **Codebeispiel 4.1**.

Codebeispiel 4.1 XML-Dokumentsicht des Knotentyps `nt:base`

```

<?xml version="1.0" encoding="UTF-8"?>
<nt:base xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  xmlns:jcr="http://www.jcp.org/jcr/1.0"
  jcr:primaryType="nt:nodeType"
  jcr:mixinTypes=""
  jcr:nodeTypeName="nt:base"
  jcr:supertypes=""
  jcr:isMixin="false"
  jcr:hasOrderableChildNodes="false">
  <jcr:propertyDefinition jcr:primaryType="nt:propertyDefinition"
    jcr:mixinTypes=""
    jcr:name="jcr:primaryType"
    jcr:autoCreated="true"
    jcr:mandatory="true"
    jcr:onParentVersion="COMPUTE"
    jcr:protected="true"
    jcr:multiple="false"
    jcr:requiredType="NAME"
    jcr:defaultValues=""
    jcr:valueConstraints="" />
  <jcr:propertyDefinition jcr:primaryType="nt:propertyDefinition"
    jcr:mixinTypes=""
    jcr:name="jcr:mixinTypes"
    jcr:autoCreated="false"
    jcr:mandatory="false"
    jcr:onParentVersion="COMPUTE"
    jcr:protected="true"
    jcr:multiple="true"
    jcr:requiredType="NAME"
    jcr:defaultValues=""
    jcr:valueConstraints="" />
</nt:base>

```

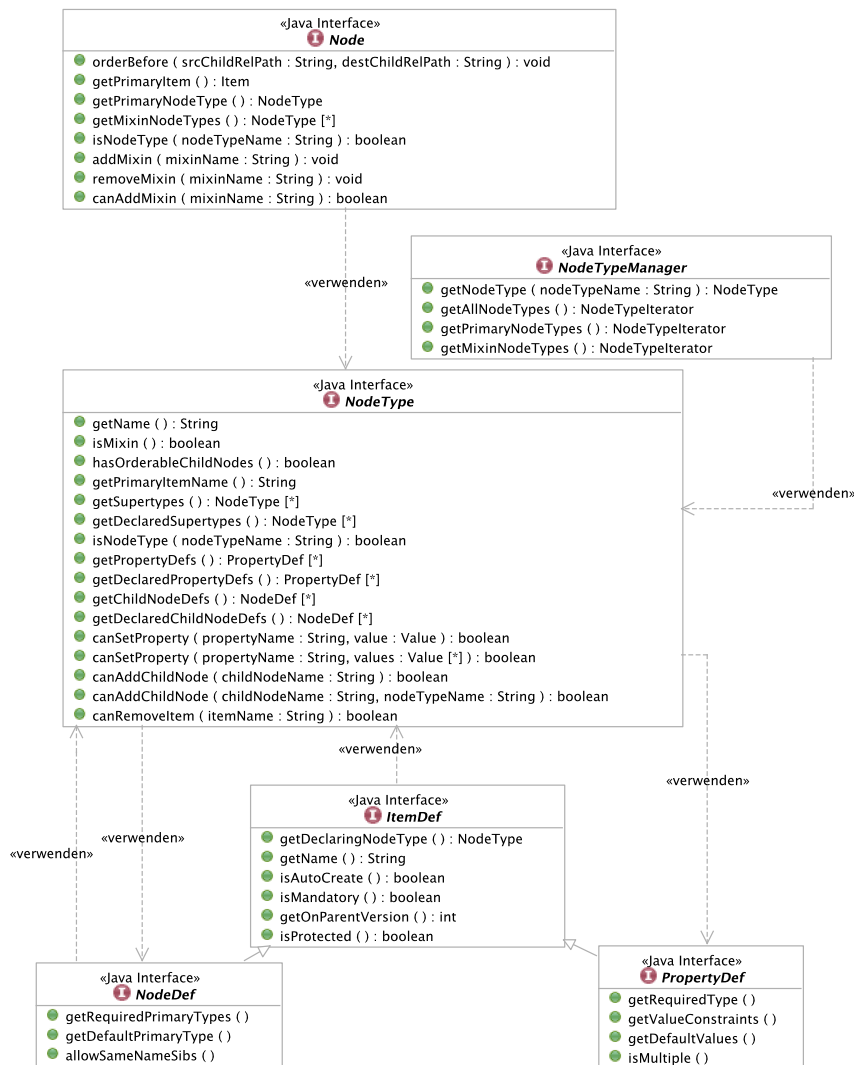


Abbildung 4.5: Klassendiagramm: JCR-Knotentypen

4.4 XML Import und Export

Für den einfachen Export und Import von Daten aus einem JCR-konformen und in ein JCR-konformes Repository steht eine XML-Schnittstelle bereit. Es handelt sich dabei um eine Abbildung der Elementhierarchie in ein XML-Dokument. Wie für alle anderen Funktionen sind in Level 1 die XML-Funktionen auf den lesenden Zugriff, also den Export von Daten, beschränkt.

Zwei Abbildungen sind für JCR definiert, die Systemsicht (*system view*) und die Dokumentsicht (*document view*).

4.4.1 Systemsicht

Für Menschen vielleicht nicht leicht verständlich, bietet die Systemsicht den 1:1-Export aller Daten des Repositories in das XML-Format. **Codebeispiel 4.2** zeigt den Export unseres Beispielarbeitsbereichs aus [Abbildung 4.2](#) in Systemsicht. Per Definition hat der Wurzelknoten eines Arbeitsbereichs einen leeren Namen („“), dieser wird sowohl in Systemsicht wie in Dokumentsicht auf den Namen `jcr:root` abgebildet. Knoten in JCR werden in `<sv:node>`-Elemente übersetzt, die als Kinder wieder `<sv:node>`-Elemente oder `<sv:property>`-Elemente, die die Eigenschaften repräsentieren, enthalten. Die Angabe `sv:type` sichert dabei, trotz Typkonvertierung in String, die Typinformation der Eigenschaft. Eigenschaftswerte sind als `<sv:value>`-Elemente innerhalb von `<sv:property>` erfasst. Mehrwertige Attribute enthalten einfach mehrere `<sv:value>`-Elemente.

Codebeispiel 4.2 Beispiel eines XML-Exports in Systemsicht

```

<?xml version="1.0" encoding="UTF-8"?>
<sv:node xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:sv="http://www.jcp.org/jcr/sv/1.0"
  sv:name="jcr:root">
  <sv:property sv:name="jcr:primaryType" sv:type="Name">
    <sv:value>nt:unstructured</sv:value>
  </sv:property>
  <sv:node sv:name="a">
    <sv:property sv:name="jcr:primaryType" sv:type="Name">
      <sv:value>nt:unstructured</sv:value>
    </sv:property>
    <sv:node sv:name="d">
      <sv:property sv:name="jcr:primaryType" sv:type="Name">
        <sv:value>nt:unstructured</sv:value>
      </sv:property>
      <sv:property sv:name="h" sv:type="Boolean">
        <sv:value>true</sv:value>
      </sv:property>
    </sv:node>
    <sv:node sv:name="e">
      <sv:property sv:name="jcr:primaryType" sv:type="Name">
        <sv:value>nt:unstructured</sv:value>
      </sv:property>
      <sv:property sv:name="i" sv:type="Binary">
        <sv:value>[binary data base64 encoded]</sv:value>
      </sv:property>
      <sv:property sv:name="j" sv:type="Double">
        <sv:value>6.022E23</sv:value>
      </sv:property>
    </sv:node>
  </sv:node>
  <sv:node sv:name="b">
    <sv:property sv:name="jcr:primaryType" sv:type="Name">
      <sv:value>nt:unstructured</sv:value>
    </sv:property>
    <sv:property sv:name="f" sv:type="String">
      <sv:value>Once upon a time...</sv:value>
    </sv:property>
  </sv:node>
  <sv:node sv:name="c">
    <sv:property sv:name="jcr:primaryType" sv:type="Name">
      <sv:value>nt:unstructured</sv:value>
    </sv:property>
    <sv:property sv:name="g" sv:type="Long">
      <sv:value>-25</sv:value>
    </sv:property>
  </sv:node>
</sv:node>

```

4.4.2 Dokumentsicht

Anders als in der Systemsicht, verzichtet man zugunsten der leichten Lesbarkeit auf einige Informationen. **Codebeispiel 4.3** zeigt den Export unseres Beispielarbeitsbereichs aus Abbildung 4.2 in Dokumentsicht. Deutlich zu erkennen, im Unterschied zu Codebeispiel 4.2, ist, dass die Werte der Eigenschaft ihren Typ verloren haben, da alle Werte in String umgewandelt worden sind und zusätzliche Typangaben, wie in der Systemsicht, hier fehlen. Sollte ein Kindknoten mit dem Namen `jcr:xmltext` mit der einzigen Eigenschaft `jcr:xmlcharacters` vorhanden sein, so wird dieser nicht als Kindelement eingefügt, sondern als Textknoten des Vaters. Sonst gilt die Regel, dass jeder Knoten ein XML-Element gleichen Namens wird und jede Eigenschaft ein Attribut dieses Elements. Die kompakte XML-Darstellung scheint jedoch tatsächlich viel lesbarer und damit leichter verarbeitbar zu sein. Sowohl Arbeitsbereich als auch XML-Dokument implizieren eine Baumstruktur. Für XML-Dokumente existiert mit XPATH eine verbreitete Abfragesprache. Um diese für JCR zu nutzen, benötigt man eine Abbildung vom JCR-Knotenmodell zu XML. Die Darstellung in Dokumentsicht orientiert sich intuitiver an dem Aufbau des Arbeitsbereichs und wurde für die XPATH-Abfragemöglichkeit (siehe Abschnitt 4.2.1) ausgewählt.

Codebeispiel 4.3 Beispiel eines XML-Exports in Dokumentsicht

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:sv="http://www.jcp.org/jcr/sv/1.0"
  jcr:primaryType="nt:unstructured">
  <a jcr:primaryType="nt:unstructured">
    <d jcr:primaryType="nt:unstructured"
      h="true"/>
    <e jcr:primaryType="nt:unstructured"
      i="[binary data base64 encoded]"
      j="6.022E23"/>
  </a>
  <b jcr:primaryType="nt:unstructured"
    f="Once upon a time..."/>
  <c jcr:primaryType="nt:unstructured"
    g="-25"/>
</jcr:root>
```

4.5 XPATH-Suche in JCR

Für Level 1 ist bereits die XPATH-Suche erforderlich (siehe 4.2.1). Das Codebeispiel 4.3 dient auch in diesem Abschnitt als Beispiel, um die JCR-Konzepte der XPATH-Suche vorzustellen. Um Implementierungen auf Datenbanksystembasis nicht zu behindern, sieht die Spezifikation allerdings nicht die vollständige Unterstützung der XPATH-Syntax vor (Nüscheler u. a. 2005, Kapitel 6.6).

4.5.1 Aufbau der Suchanfrage

Ergebnis einer Suchanfrage (*result set*) ist eine Menge von Knoten. Das Ergebnis ist eine Untermenge von Knoten des Arbeitsbereiches, auf denen bestimmte Einschränkungen (die der Suche) zutreffen. Laut Spezifikation gibt es drei Fälle solcher Einschränkungen.

- **Typeinschränkung.** Dies limitiert die zurückgegebene Knotenmenge auf Knoten eines bestimmten primären Knotentyps. Optional ist die Limitierung auf bestimmte „mixin“ Knotentypen. Knotentypentests sind vererbungssicher. Das bedeutet, dass bei Einschränkung auf einen Typ *x* alle zurückgegebenen Knoten entweder von Typ *x* oder von einem Subtyp von *x* sind. Der Test zur Typeinschränkung lautet `element()` und kann entweder am Anfang oder am Ende des XPATH-Ausdrucks stehen:

- `//element(*, nt:base)` liefert alle Knoten zurück.
- `//element(metadata, *)` liefert alle Knoten mit den Namen `metadata` zurück.

- **Eigenschaftseinschränkung.** Dies limitiert die zurückgegebene Knotenmenge auf Knoten mit bestimmten Eigenschaften und Eigenschaftswerten. Am Ende des XPATH-Ausdrucks können die Knoten mittels Prädikat nach Eigenschaften gefiltert werden. Unterstützung für Eigenschaftstest am Anfang oder der Mitte des Ausdrucks ist für JCR-Konformität nicht erforderlich. Der Grund dafür ist vermutlich in der Nutzung von Indizes zu suchen, die nur diese Art von Anfragen effektiv beantworten können.

- `//element(*, nt:base)[@title='JCR']` liefert alle (vom Typ `nt:base`) Knoten zurück, die eine Eigenschaft `title` mit Wert `JCR` haben.
- `//element(*, mytype:employee)[@manager and @salary>10000]` liefert alle Knoten vom Typ `mytype:employee`,

die die Eigenschaft `manager` besitzen und für die Eigenschaft `salary` einen Wert von über 10000.

- **Pfadeinschränkung.** Dies limitiert die zurückgegebenen Werte auf bestimmte Teilbäume des Arbeitsbereichs. Folgende Einschränkungen, die nicht dem gesamten XPATH-Umfang entsprechen, müssen unterstützt werden:

- Exakt

- * `/jcr:root/path/to/nodes/element(nodes, my:type)` liefert alle Knoten mit Namen `nodes` vom Typ `my:type` unterhalb des Teilbaums `/jcr:root/path/to/nodes`.

- Kindknoten

- * `/jcr:root/path/to/nodes/element(*, my:type)` liefert alle Kinderknoten von `nodes` vom Typ `my:type` unterhalb des Teilbaums `/jcr:root/path/to`.

- Nachfahren

- * `/jcr:root/path/to/nodes//element(*, my:type)` liefert alle Nachfolgeknoten (Kinder, Kindeskind, etc.) von `nodes` vom Typ `my:type` unterhalb des Teilbaums `/jcr:root/path/to`.

4.5.2 Einschränkungen gegenüber XPATH

Anders als mit vollem XPATH-Umfang, wird grundsätzlich nur die abkürzende Schreibweise verlangt, auch die Unterstützung der Vorfahrachse wird nicht verlangt. Das Prädikat ist auf Test von Attributen (Eigenschaften) beschränkt. Diese Eingrenzungen machen folgende Suchanfragen unmöglich:

- `/jcr:root/path/to/nodes/child::` äquivalent zur unterstützten Syntax `/jcr:root/path/to/nodes/element(*, *)`
- `/jcr:root/document[metadata/title/@main="JCR"]` würde alle `document`-Knoten liefern, die im Kind `metadata` ein Element `title` mit dem Wert 'JCR' im Attribut `main` haben. Im Unterschied dazu würde die Anfrage `/jcr:root/document/metadata/title[@main="JCR"]` `title`-Knoten liefern.

- **`//*[jcr:contains(., 'JSR 170')]/ancestor::myApp:node`**
würde alle Knoten liefern, die eine Eigenschaft mit dem Wert 'JSR 170' besitzen und zusätzlich als Vorfahren (Vater, Großvater, etc.) einen Knoten mit dem Namen `myApp:node`. Die Funktion `(jcr:contains())` ist eine JCR XPATH-Erweiterung zur Volltextsuche.

4.5.3 Erweiterungen gegenüber XPATH

Neben den eben erwähnten Einschränkungen, existieren auch eine Reihe von Erweiterungen der XPATH-Syntax. Dies sind vor allem Funktionen, aber auch eine Möglichkeit, die Rückgabemenge zu sortieren. Auch hier wird oft deutlich, wie sehr auf ein mögliches Datenbank-Backend Rücksicht genommen wird, es seine Stärken somit auch ausspielen kann.

- **Die `jcr:like()` Funktion** basiert auf dem `LIKE`-Operator von SQL und ihre Unterstützung ist erforderlich. Zum Beispiel findet die Anfrage `//*[jcr:like(title, 'JSR-%')]` alle Knoten, die eine Eigenschaft `title` haben, deren Wert mit 'JSR-' beginnt. Als Platzhalter gilt neben dem erwähnten Prozentzeichen („%“) für beliebige Zeichenketten der Unterstrich („_“), der für einzelne Buchstaben steht. Ihre regulären Vorkommen in Zeichenketten innerhalb der `jcr:like`-Funktion müssen mit einem Backslash („\") maskiert werden.
- **Die `jcr:contains()` Funktion** stellt, wie bereits erwähnt, eine Volltextsuche bereit und basiert wiederum auf der SQL Funktion `CONTAINS`. Sie benötigt zwei Parameter, der erste grenzt den Suchbereich ein, z.B. auf eine bestimmte Eigenschaft. Jedoch ist dies optional und nur „_“-Unterstützung erforderlich, was für „alle Eigenschaften“ steht. Der zweite Parameter ist der Suchtext. So sucht `//*[jcr:contains(., JSR-JSR-170)]` nach allen Knoten, in deren Eigenschaften das Wort 'JSR', jedoch nicht 'JSR-170' vorkommt.
- **Die `jcr:deref()` Funktion** folgt einer Eigenschaft vom Typ `REFERENCE` zum Ziel. Ihre Unterstützung ist optional. So findet zum Beispiel `/jcr:root/myapp:myDoc/jcr:deref(@myapp:author, 'myapp:person')/address` die Adressknoten (`address`) von Personen `myapp:person`, die als Autor `@myapp:author` für Dokumente `myapp:myDoc` eingetragen sind.
- **Die `order by` Klausel** sortiert die zurückgegebenen Knoten nach einem oder einer Reihe von Eigenschaftswerten. Zum Beispiel sortiert `/jcr:root/myapp:myDoc order by @title descending` alle `myapp:myDoc`-Knoten in der Ergebnismenge absteigend nach dem

Wert der `title`-Eigenschaft. Als Sortierkriterium sind ausschließlich Eigenschaftswerte des sortierten Knotens zulässig, nicht jedoch Eigenschaftswerte von Nachfolgerknoten, wie zum Beispiel bei `/jcr:root/document order by metadata/mytitle/@main`.

Die erwähnten Funktionen erweitern die XPATH-Abfragesprache um JCR-typische Funktionalitäten sinnvoll, die aus diesem Grund nicht Teil der XPATH-Sprache sind. Sie gestatten den Zugriff auf Volltextindizes und auf spezielle Eigenschaften von JCR-Knoten.

4.6 Fazit

Als reine Content-Repository-API entworfen, bietet JCR keine typischen Funktionen eines Content-Management-Systems, wie Nutzerverwaltung, User-Interfaces oder Workflow-Unterstützung. Das war jedoch auch nicht das Ziel der Entwicklung, sondern die API-Definition eines Datenspeichers. Die vorliegende Spezifikation gibt, von den Unstimmigkeiten bei Zugriffskontrolle (siehe Seite 57) abgesehen, ein rundes Bild ab, was typische Funktionen eines Content-Repositories angeht. *Spezialisierung*

Es ermöglicht eine schnelle Adaptierung an vorhandene Systeme, durch die Gliederung in unterschiedliche Level der Konformität (siehe Abschnitt 4.2.1, 4.2.2 und 4.2.3). Es ist zur Zeit noch nicht möglich, die endgültige Unterstützung der Schnittstelle durch die Hersteller abzuschätzen. Die Vermutung liegt aber nahe, dass die Unternehmen und Organisatoren, die in der Mitgliederliste der JSR-170 Expert-Group aufgelistet sind, ihr Mitwirken in der Spezifikation in Produkte umsetzen werden. So bestätigten neben Day Software auch FileNet und IBM bereits an Produkten zu arbeiten oder diese schon anzubieten (Day Software 2005a) *PRO*

„The standard will have a major impact on the content management landscape, separating infrastructural services from application services as has long been prevalent in the data world.“

David Nüscheler (Day Software 2005b)

Hinderlich an der Akzeptanz der Schnittstelle könnten sich jedoch die „Löcher“ in der Spezifikation herausstellen. Dies betrifft nicht die vielen optionalen Funktionen und Definitionen. Vielmehr fehlt es an Regelungen, wie Anwendungsentwickler Knotentypen herstellerübergreifend definieren können. Sie sind diesbezüglich entweder auf eine, hoffentlich mögliche, herstellerübergreifende Spezifikation angewiesen oder arbeiten wieder mit einer herstellerspezifischen API, sollte diese zur Verfügung gestellt werden. Letzteres wollte man eigentlich vermeiden, beschränkt es aber damit zumindest auf ein Minimum. *CONTRA*

„JCR wird sicher über die v1.0 innerhalb des JCP weiterentwickelt. Dafür wird es einen neuen JSR geben. Ich halte die Zeit für reif über diese Punkte erneut zu diskutieren und vielleicht haben wir in JCR 1.1 oder 2.0 eine Lösung für die obigen Punkte. [Jackrabbit]⁷ hat Vorschläge für eine zukünftige mögliche Standardlösung. Eine kleine UI-Demo gibts auch unter: [JCR Seite von Day]⁸ (content explorer / security)“

⁷<http://incubator.apache.org/jackrabbit>

⁸<http://jcr.day.com>

(David Nüscheler über Anbindung von Zugriffskontrollsystemen und Knotentypdefinition in JCR in einer Email vom 24.06.2005)

Es blickt an vielen Stellen durch, dass allzu sehr die Unterstützung von SQL-Datenbanksystemen als Backend im Vordergrund stand. So wurde in Abschnitt 4.5.2 gezeigt, auf welche praktischen Abfragemöglichkeiten der XPATH-Spezifikation man verzichten muss zugunsten der Datenbankunterstützung.

„Therefore, in order to ensure that database-backend implementations are not unnecessarily burdened by compliance requirement, only a subset of XPath is required. This subset is defined as the set of XPath statements that can be translated to and from SQL at parse-time of the query.“

(Nüscheler u. a. 2005, Seite 97)

So sehr die unterschiedlichen Levels der Konformität eine rasche Implementierung ermöglichen, so hinderlich sind sie bei dem Austauschen von Repositories untereinander. Es wird, zumindest anfangs, schwer fallen, Repositories mit gleichem Funktionsumfang zu finden. Es ist jedoch Sache der Hersteller diese Phase möglichst kurz zu halten. Allerdings lassen sich, bis auf den Export der Daten, der grundsätzlich gewährleistet wird, darüber hinaus an dem Prädikat „JCR-kompatibel“ keine qualitativen Aussagen über ein Produkt treffen.

Quintessenz

Unterm Strich bleibt das Fazit, dass JCR ein Schritt in eine Richtung ist, die Anwendungsentwicklern die Arbeit erleichtern kann. Die Spezifikation nimmt viel Arbeit ab und wird sicherlich langfristig dazu führen, dass Content-Repositories und Content-Management nicht mehr synonym behandelt, sondern stärker zwischen ihnen unterschieden wird. Es gilt jedoch für Content-Management-Systeme weiterhin genau zu untersuchen, ob JCR neben der zweifellosen Erleichterung in der Programmierung schon jetzt die Funktionen bereitstellt, die in der Praxis gewünscht werden. In erster Linie fallen da die erwähnten Probleme mit der XPATH-Anfrage ein, den Export und Import von Knotentypdefinitionen und die Anbindung von Zugriffskontrollsystemen, die bisher noch nicht gelöst worden sind.

Kapitel 5

Vergleich der Architekturmodelle

5.1 Vergleich der Persistenzfunktionen

In den Kapiteln 3 und 4 wurden die beiden Standards MyCoRe und JCR untersucht. Bei der abschließenden Beurteilung muss man berücksichtigen, dass beide zwar gemeinsame Funktionalität haben, jedoch aus unterschiedlichen Gründen entwickelt worden sind. JCR ist als reine Schnittstelle für Repositories entstanden, die die wichtigsten und häufigsten Funktionen standardisiert zugreifbar machen soll. Das Funktionsspektrum von MyCoRe hingegen ist mehr dem eines Content-Management-Systems. Da täuscht der Name „My Content Repository“. Es bietet Schnittstellen zu Webanwendungen und anderen Systemen, stellt aber auch eine Repository-Schnittstelle bereit, die ausschließlich Gegenstand dieser Arbeit war.

MyCoRe: mehr als ein Repository

Tabelle 5.1 dient als kurze Zusammenfassung der folgenden Abschnitte, in denen der Funktionsumfang diskutiert wird. Jede Funktion wird dort bewertet. Dabei stehen „+“ beziehungsweise „++“ für Funktionen, die unterstützt beziehungsweise im besonderen Maße unterstützt werden. Dem gegenüber sind „–“ und „––“ Funktionen, die fehlen oder konzeptionell nicht oder nur schwer realisierbar sind. Ein Wert von „0“ steht für Funktionen, die vorhanden, aber unzureichend implementiert oder spezifiziert worden sind.

Beide Lösungskonzepte bieten die Grundfunktionalitäten wie Schreib- und Lesezugriffe an, ohne das dabei das eine System das andere in diesen Punkten ausstechen könnte. Zugriffsbeschränkungen unterstützt von beiden nur JCR und das auch nur in grundlegenden Funktionen. Es orientiert sich an dem Aktionenkonzept der JAAS-Spezifikation (vgl. Abschnitt 4.2.1). Es ist daher

Grundfunktionen

prinzipbedingt recht generisch, jedoch nicht von bestimmten Ausprägungen der Zugriffskontrolle wie ACL abhängig, was durchaus vorteilhaft sein kann.

Integrität

In Abschnitt 3.6.1 wurden die Schwächen von MyCoRe bezüglich der referentiellen Integrität beschrieben. Dies ist jedoch keine Funktion, die prinzipbedingt fehlt. MyCoRe wäre in der Lage, die referentielle Integrität sicherzustellen, jedoch wurde bei der Implementierung dieser Punkt vernachlässigt. Über die Implementierung eines JCR-Repositories kann hier natürlich keine Beurteilung erfolgen. Jedoch beschreibt die Spezifikation sehr detailliert, wie referentielle Integrität sichergestellt wird und so sei angenommen, dass ein Content-Repository den JCR-Standard vollständig umgesetzt hat.

Metadaten-Funktionen

Ein Vorteil von MyCoRe sind die eigenen Datentypen (vgl. Abschnitt 3.2). Es wird die Möglichkeit gegeben, spezielle Logik zur Integritätssicherung in eigene Klassen zu integrieren, um „beliebige“ Sachverhalte zu überprüfen, z.B. die Gültigkeit von ISBN-Nummern. MyCoRe erlaubt es ferner, dass Kinder Metadaten von ihrem Vater erben können. Diese Möglichkeiten sind so weitgreifend von JCR nicht vorgesehen. Man baut auf Knotentypen (vgl. Abschnitt 4.3), die grundlegende Datentypen, wie Strings, Fest- und Gleitkommazahlen fußen.

XML-Export

Sowohl JCR als auch MyCoRe unterstützen den Import und Export von XML-Daten. Durch die Möglichkeit von MyCoRe, für den Export von XML-Daten beliebige Stylesheets zu verwenden, sind die Anwendungsbeispiele natürlich sehr breit gefächert. Eine Möglichkeit, diese Funktionen zu nutzen, ist, ganze Webapplikationen zu bauen, die nur auf dem XML-Export von MyCoRe aufgebaut sind und durch XSLT eine Weboberfläche dem Anwender zur Verfügung stellen. In diesem Zusammenhang wird deutlich, dass hier die Funktionen die eines reinen Content-Repositories übersteigen. Die starke Importfunktion von JCR, die beliebiges XML importieren und verwalten kann (vgl. Abschnitt 4.4), ist so in MyCoRe hingegen gar nicht möglich.

Eine weitere Schwäche bezüglich MyCoRe ist die Namespace-Unterstützung, die bei der Entwicklung nicht berücksichtigt wurde – XLINKS ausgenommen. Jedoch sind gerade die Funktionen im Bereich Namensräume bei JCR sehr weit gediehen (vgl. Abschnitte 4.2.1 und 4.2.2).

Suche im Repository

Die Suchmöglichkeiten sind bei beiden „Kontrahenten“ vielfältig. Hier kann sich abwechselnd der eine vom anderen absetzen. Die schon nicht umfassende XPATH-Unterstützung von MyCoRe wird, wie in Abschnitt 4.5 dargestellt, nur noch von JCR unterboten. Beide stellen auf der anderen Seite ähnliche Funktionen bereit, die über die der reinen XPATH-Syntax hinaus gehen, um zum Beispiel Volltextsuchen bereitzustellen. Letzere wird in JCR nur bei reinem Text unterstützt, während diesbezüglich MyCoRe nicht limitiert ist, um Volltexte von Formaten wie PDF oder Word zu indizieren. Durch die direkte Standardisierung von SQL als Abfragesprache existiert bei JCR eine Alternative zur

XPATH-Syntax. Durch `QueryManager.getSupportedQueryLanguages()` sind noch eine Vielzahl anderer Abfragesprachen möglich, deren Registrierung im Repository ist jedoch nicht spezifiziert.

Darüber hinausgehende Funktionen, wie sie JCR bezüglich Transaktionen, Versionierung und Sperren bietet sind von Grund auf schwer in MyCoRe zu integrieren, weil die Softwarearchitektur nicht strikt getrennt ist (vgl. Abschnitt 3.6). Neben dem erhöhten Aufwand, ist aus diesem Grund mit zahlreichen Seiteneffekten zu rechnen.

Die in MyCoRe 1.0 fehlende Ereigniskontrolle wurde in die Version 1.1 integriert, die während der Entstehung dieser Diplomarbeit veröffentlicht worden ist.

Funktionen der Persistenz von JCR und MyCoRe		
Funktion	MyCoRe 1.0	JCR 1.0
Lesezugriff	+	+
Schreibzugriff	+	+
Zugriffsbeschränkung	–	+
referentielle Integrität	o	+
eigene Daten-/Dokumenttypen	++	+
XML-Export	++	+
XML-Import	+	++
XML-Schemasupport	+	o
Unterstützung Namensräume	--	++
verteilte Suche	+	--
XPATH-Suche	+	o
XPATH erweitert	+	+
Volltextsuche	+	+
Vererbung	+	–
andere Abfragesprachen	--	+
Transaktionen	--	+
Versionierung	--	+
Sperren	--	+
Ereigniskontrolle	–	+

Tabelle 5.1: Funktionsmatrix: Persistenz von MyCoRe und JCR

Abschließend bleibt nur noch einmal herauszustellen, dass sowohl MyCoRe wie auch JCR in der Version 1.0 untersucht worden sind. Man kann in beiden

Fällen ausgehen, dass Schwächen der ersten „stabilen“ Version in zukünftigen Versionen nicht mehr so in Erscheinung treten werden.

5.2 Vergleich der Architekturen

System vs. API

Bislang wurde untersucht, welche Funktionen von welchem System bereitgestellt werden und wie diese angeboten werden. Noch nicht betrachtet wurde bislang, wie sich die MyCoRe-Repository-Schnittstelle von der JCR-Schnittstelle unterscheidet. Augenmerklichster Unterschied ist, dass JCR nur eine Sammlung von Schnittstellen (*Interfaces*) ist, während MyCoRe einen Großteil der Funktionen implementiert. Zur Folge hat dies, dass es keine eigenen Implementierungen eines MyCoRe-Repositories geben kann, weil nur ein Teil der Funktionalität über Schnittstellen angeboten wird. Intern werden eine Vielzahl davon von MyCoRe genutzt, um unterschiedliche Funktionen „herstellerunabhängig“ ansprechen zu können. Dies gelingt auch, setzt aber voraus, dass die Arbeit über diesen Schnittstellen gering ist, um effektiv arbeiten zu können. Effektiv meint hier, dass Arbeitsschritte von spezialisierten Komponenten übernommen werden sollten, wenn dies möglich ist, um die Ausführungszeit zu verkürzen. Die Aufteilung sollte es also trotz Teilung ermöglichen, dass die Hauptlast der Arbeit nicht bei MyCoRe liegt, sondern bei den angesteuerten Komponenten. Dies verkürzt nicht nur die Ausführungszeit, sondern verringert auch den selbst zu wartenden Code.

Nun dokumentierte Kapitel 3, dass die Hauptarbeit oberhalb dieser Schnittstellen erfolgt. Dies zeigte sich unter anderem beim Löschen und Einstellen von MyCoRe-Objekten. Auch das Trennen der Persistenzschnittstellen in Speicher für Metadaten, Suchindizes und Dateien trägt dazu bei, dass andere, MyCoRe-fremde Produkte nicht mit diesen Daten arbeiten können. So ist es nicht möglich, sämtliche Daten an ein Repository zu übergeben und die Suche über seine Schnittstellen durchzuführen. Bei JCR sieht der Ansatz anders aus. Hier verständigte man sich auf ein gemeinsames Architekturmodell und auf einen gemeinsamen Funktionsumfang. Jedem Anbieter von Zusatzfunktionen für Repositories ist es auf Grund dieser gemeinsamen Spezifikation möglich, diese herstellerübergreifend anzubieten. Zum Beispiel könnten so Hot-Backup¹-Funktionen in ein Repository integriert werden. Allerdings erlaubt der Ansatz auch, dass Repository-Hersteller eigene zusätzliche Funktionen anbieten können, die unabhängig vom Anwendungsprogramm funktionieren, welches auf JCR aufsetzt.

Unterschiede in Anforderungen

Generell lässt sich sagen, dass MyCoRe möglichst wenig Anforderungen an das Repository-Backend stellt, dafür aber dem Backend-Anbieter nicht die

¹Replikation mit möglichst aktuellen Daten des Live-Systems. Bei Ausfall des Live-Systems kann ein Hot-Standby-Server seine Aufgaben (mit „fast“ identischem Datenstand) übernehmen.

Möglichkeit gibt, seine Stärken, sowohl funktionell, wie auch hoher Geschwindigkeit ablaufend, zur Verfügung zu stellen. JCR staffelt diese Anforderungen in unterschiedliche Level der Konformität. So werden einfache Implementierungen möglich, aber auch eine Großteil von Repository-Funktionen mittels einer API zugänglich. Das Gewicht der Logik ist bei MyCoRe eher anwendungsseitig zu sehen, während JCR eher dem Repository spezifische Funktionen überträgt.

Man sieht, dass obwohl die Funktionen, die von beiden Systemen bereitgestellt werden (vgl. Tabelle 5.1), sich ähneln, relativ unterschiedliche Architekturkonzepte zum Tragen kamen. Das JCR-Konzept stellt sich aber insgesamt in sich geschlossener dar als die Persistenzschnittstellen von MyCoRe. Aufgrund der teilweise großen Überdeckung von JCR in MyCoRe bleibt noch eine Frage offen: Lohnt sich die Integration von JCR in MyCoRe? Dies versucht das folgende Kapitel zu klären.

Kapitel 6

Architekturentwurf für ein MyCoRe *Next*

Das Kapitel wurde etwas provokativ „Architekturentwurf für ein MyCoRe *Next*“ genannt. Hier sollen noch einmal mögliche Schlussfolgerungen aus den vorangegangenen Kapiteln aufgezeigt werden. Natürlich kann man im Rahmen einer Diplomarbeit, wie sie hier vorliegt, auch keinen vollständigen Architekturentwurf machen. Es gilt einerseits, wenn möglich, die Kompatibilität zu beachten und andererseits das neue System nicht nur schneller und funktionsreicher zu machen, sondern auch verständlicher und damit leichter wartbar.

In diesem Kapitel wird im ersten Schritt noch einmal kurz diskutiert, warum MyCoRe von einer JCR-Unterstützung profitieren kann. Die momentanen Grenzen werden ebenso verdeutlicht. Die Beurteilung hierzu beruht auf den Fakten von Kapitel 3 und 4.

In einem weiteren Abschnitt werden mögliche Abbildungen des MyCoRe-Datenmodells auf JCR vorgestellt und welche Bedingungen an die jeweiligen Vorschläge geknüpft werden müssen. Je mehr Änderungen man letztendlich bereit ist einzugehen, desto mehr Möglichkeiten von JCR stehen offen zur Verfügung.

Abschließend werden Konzepte vorgestellt, wie die Architektur von MyCoRe generell angepasst werden muss, um zukünftig Aufgaben wie Wartung und Entwicklung zu beschleunigen und Fehlerquellen zu reduzieren.

6.1 Diskussion: MyCoRe auf JCR?

In Abschnitt 3.6 wurde deutlich, dass es MyCoRe vor allem an der Umsetzung eines genau definierten Architekturkonzeptes mangelt. Dies betrifft sowohl

die Datenmodellschicht als auch die Persistenzschicht, die in Abschnitt 3.1 vorgestellt worden sind. Die Verschränkung dieser beide Ebenen zu lösen, wird man es nicht verhindern können in einen Neuentwurf mit einzubeziehen. Natürlich bietet es sich an, zu einen frühen Zeitpunkt alle ursprünglichen Überlegungen noch einmal abzuwägen. Ausgehend von Anforderungen, die in der Praxis eine Rolle spielen (vgl. Kapitel 2), kann das System dann einem neuen Design unterworfen werden.

Im Zuge dieser Änderungen und auf Grund der gezeigten Schwächen sollte man versuchen Code in gut gepflegte Software-Projekte auszulagern, um den eigenen Codestand zu verringern. Dies allein kann schon dafür sorgen, dass der Code leichter wartbar wird und damit weniger Fehlerquellen beherbergt. Auch zwingt dieses Vorgehen zum Einhalten von Schnittstellen, da man den fremden Code in der Regel nicht direkt beeinflussen kann.

In Kapitel 4 zeigte sich, dass eine Methode bereitsteht, die jegliche Persistenzaufgaben übernehmen kann und besonders Funktionen bereitstellt, die in der Form nur schwer in MyCoRe zu realisieren sind. Es wurde jedoch deutlich, dass JCR in der Version 1.0 (JSR-170) noch einige Schwächen bezüglich der Abfragemöglichkeiten und in der Anbindung von Zugriffskontrollsystemen hat. Letzteres sind aber Schwächen, die sich nur beim Setzen von Zugriffsrechten zeigen. Bedingt dadurch gibt es auch keine standardisierten Export- und Import-Regeln für oft sehr umfangreiche Zugriffsregelungen. Sollten diese Punkte zu den Anforderungen von MyCoRe gehören, so muss man trotzdem nicht auf die Vorzüge von JCR verzichten. JCR wird nach Worten des *Specification Leaders* David Nüscheler schon bald in diese Richtung weiter entwickelt. Bis also diese Probleme behoben werden, kann man Vorkehrungen treffen, um ca. 95% der Persistenzschicht auf diesem Standard zu bauen und die restlichen (geschätzten) 5% in eigener Regie zu entwickeln. Die daraus entstehende Architektur zeigt **Abbildung 6.3**. Später müssen dann nur noch diese letzten Prozentpunkte des Codes angepasst werden, um auch diese letzten Funktionen auszulagern.

Einen Punkt zeigt **Abbildung 6.3** im Unterschied zu **Abbildung 3.1** jedoch auch. Es wird nicht mehr unterschieden, ob Daten in einer XML:DB, dem Content Manager oder dem Dateisystem gespeichert werden. Dies entscheidet nun der Repository-Anbieter. Allerdings stellt sich die Frage nach der Volltextindizierung von Daten, die nicht im reinen Textformat vorliegen. Es fällt zur Zeit schwer, zu beantworten, wie dieses Problem zu lösen sein kann. Der generische Weg führt über die *Observer*-Funktionalität von JCR, die beim Einfügen von Knoten zusätzliche Funktionen ausführen kann. Über diese „Brücke“ kann man auch einer selbstgeschriebenen, vollständigen XPATH-Suchkomponente die Möglichkeit geben, einen eigenen Index aufzubauen. Dieses Vorgehen kostet jedoch zusätzlichen Aufwand zur Laufzeit.

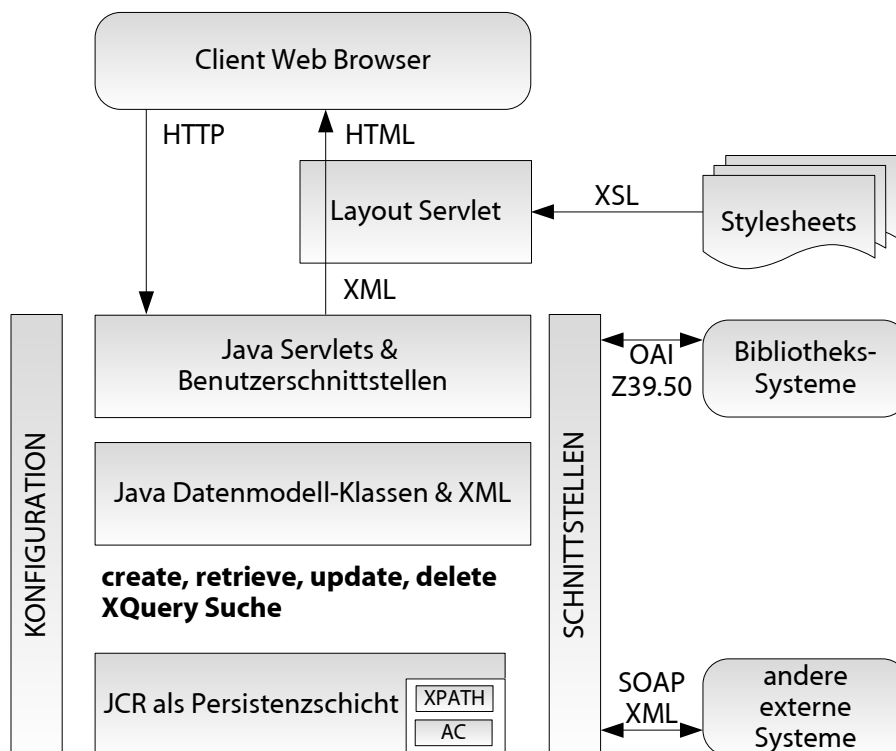


Abbildung 6.1: Architektur des MyCoRe auf JCR

Die bisherige Aufteilung in unterschiedliche Backends würde in eine eigenen JCR-konformen Implementierung der Persistenzschicht von MyCoRe 1.0 weiterleben können. Hier muss man aber den nicht unerheblichen Portierungsaufwand beachten und die Grenzen, die sich aufzeigen. So gibt es über diesem Weg nicht automatisch Versionierung und Transaktionen. Allerdings kann auch die Persistenzschicht von MyCoRe aufgrund der relativ niedrigen Hürden von JCR als Level-2-Repository dienen. Es gilt in den folgende Abschnitten deshalb auch, Informationen zu sammeln, in welchem Umfang bestimmte Funktionen der optionalen Teile der JCR-Spezifikation benötigt werden und wo die Anforderungen auch über diese hinaus gehen.

6.2 Abbildung des MyCoRe-Datenmodels auf JCR

Auch wenn es bislang noch nicht explizit erwähnt wurde: Das Abbilden von hierarchischen Modellen auf ein JCR-Repository ist keine eindeutige Sache. Als Beispiel für mögliche Portierungen wird eine Dokumentenhierarchie ange-

nommen, die in **Abbildung 6.2** dargestellt wird. Die Grafik verdeutlicht die MyCoRe-Sicht der Dokumentenbeziehungen.

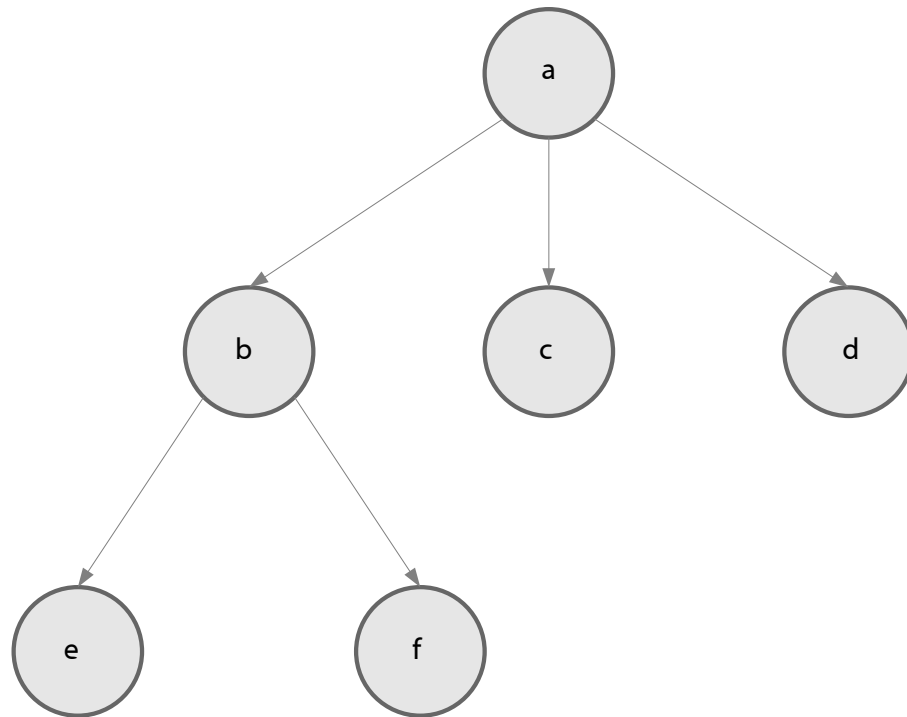


Abbildung 6.2: Logische Hierarchie von MyCoRe-Dokumenten

6.2.1 Portierung mit geringstmöglicher Änderung

Die in diesem Abschnitt beschriebene Portierung nutzt die XML-Import Funktionen für die Dokumentsicht von JCR aus (vgl. Abschnitt 4.4.2). Da MyCoRes XML-Dokumente nicht nur aus dem Wurzelement und Attributen bestehen, sondern eine echte, nicht eine flach entartete, Hierarchie bilden, kommen grundsätzlich nur Repositories in Frage, die dies auch unterstützen. Es sei nochmal daran erinnert, dass allein `nt:base` als Knotentyp unterstützt werden muss, der keinerlei Hierarchie unterstützt. Dies wird also im Fall von MyCoRe nicht ausreichen. Mit `nt:unstructured` (vgl. **Tabelle B.1**) existiert ein sehr flexibler Knotentyp. Er unterstützt *Same Name Siblings* bei Kindknoten, das heißt andere Knoten die einen gemeinsamen Vater haben, teilen den gleichen Namen. Zusätzlich ist die Anordnung der Kindknoten vom Client aus steuerbar (vgl. Abschnitt 4.3.2). Funktionen wie mehrwertige Attribute gehen über den Funktionsbedarf von MyCoRe hinaus. Negative Auswirkungen hat dies nicht.

In diesem Zusammenhang fällt jedoch ein anderer Teil der JCR-Spezifikation negativ ins Gewicht.

„In document view serializion, if the property being serialized is mutli-valued (or if the implementation chooses to encode spaces in single value properties too [...]) then the value or values must be further encoded by escaping any occurence of one of the four whitespace characters: space, tab, carriage return and line feed [...] to the following: space becomes `_x0020_`, a tab becomes `_x0009_`, a carriage return becomes `_x000D_` and a line feed becomes `_x000A_` and any underscore (`_`) that could be misinterpreted as an escape sequence becomes `_x005f_`.“

(Nüscheler u. a. 2005, Kapitel 6.4.4, Seite 91)

Das bedeutet, das zwar bedenkenlos jedes XML-Dokument eingestellt werden kann. Es wird jedoch nicht unbedingt in der Form beim Export wiederhergestellt werden können, ohne dass ein zusätzlicher Bearbeitungsschritt diese Aufgabe erledigt. Dieses Problem ist ausschließlich auf die XML-Attribute beschränkt, so dass man mit einem einfachen zwischengeschalteten XSLT-Stylesheet das Problem lösen kann. Die beschränkte Anzahl an Escape-Sequenzen (fünf in der Zahl) ist zudem leicht überschaubar. Diese Lösung ist dann zu bevorzugen, wenn man das neue System zu MyCoRe 1.0 möglichst kompatibel halten möchte. Verzichten muss man dabei auf die referenzielle Integrität von Seiten des Repositories, die nur unterstützt wird, wenn man die UUID (vgl. Abschnitt 4.2.1) ins Konzept mit einbezieht. Dieser Weg wird von der nächsten Lösung bestritten. Einfachste Integration

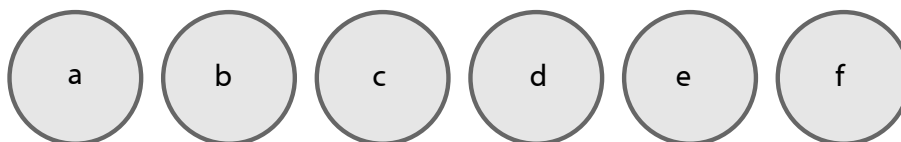


Abbildung 6.3: JCR-Sicht der Dokumenten-Hierarchie (Model 1)

6.2.2 Portierung mit Nutzung der UUID

Der eben dargestellte Ansatz verzichtet aus Gründen der Kompatibilität auf zusätzlich unterstützende Funktionen von JCR. Dazu gehört auch die Sicherstellung der referentiellen Integrität. Sind Knoten vom Typ `mix:referenceable` (vgl. **Tabelle B.2**), so ist ein JCR Repository verpflichtet, diesen Knoten erst dann zu löschen, wenn keine Referenzen auf ihn existieren. Das Problem an den

UUID und JCR ist, dass beim Anlegen von Knoten die UUID vom System, dem Repository, zugewiesen wird. Es ist daher nicht möglich, einfach die MyCoRe-Objekt-ID (siehe Abschnitt 3.2) als UUID zu verwenden. Man kann natürlich weiterhin eine MyCoRe-ID (MCRID) MyCoRe-intern verwenden. Es gäbe dann eine 1 : 1-Beziehung zwischen der MCRID und der UUID. Der Zugriff auf ein MyCoRe-Objekt über die MCRID könnte je nach JCR-Implementierung jedoch teurer sein als über die UUID. Den Zugriff über UUID erlaubt die Methode `Session.getNodeByUUID`. Der Zugriff über die MCRID erfolgt jedoch über eine XPATH-Suche (z.B. `//mycoreobject[@ID='DiplArb_sample_1']`), die in der Regel nicht so schnell beantwortet werden kann.

Gewinn referentieller Integrität

Eine Neuauslegung der Persistenzschicht könnte die Generierung der ID der Persistenz-Schicht überlassen, um so von referentieller Integrität durch das JCR-Repository zu profitieren bei gleichzeitig hoher Zugriffsgeschwindigkeit.

Diese Lösung setzt aber voraus, dass man den Dokumenttyp feststellen kann, was bislang über die MCRID geschieht. Hier bietet es sich an Dokumenttypen auf JCR-Knotentypen abzubilden. Da es keine vordefinierten Knotentypen für MyCoRe gibt, ist man zwingend auf ein Repository angewiesen, was eigene definierte Knotentypen erlaubt. Eine Möglichkeit der Definition von Knotentypen wurde in Abschnitt 4.3 auf Seite 65 gezeigt.

Tabelle 6.1 zeigt beispielhaft die Knotendefinition eines MyCoRe-Objekts. Diese Definition kann Supertyp beliebiger Ausprägungen von Dokumenttypen (z.B. „sample“) sein. Das `<structure>`-Element (vgl. Abschnitt 3.2) taucht in Form der Eigenschaften `mcr:parent` und `mcr:child` auf. Letztgenannte Eigenschaft kann per Definition mehrfach vorkommen, während dies für `mcr:parent` natürlich nicht zutrifft. Mit diesen Eigenschaften können wir nun unsere Hierarchie von MyCoRe-Objekten modellieren.

mcrnt:object	
Name	Wert
NodeTypename	mcrnt:object
Supertypes	nt:base
IsMixin	false
HasOrderableChildNodes	false
PrimaryItemName	mcr:metadata
ChildNodeDefinition	
Name	mcr:metadata
RequiredPrimaryTypes	[mcrnt:metadata]
DefaultPrimaryType	mcrnt:metadata

Fortsetzung auf nächster Seite

Tabelle 6.1 – Fortsetzung von voriger Seite

Name	Wert
<i>AutoCreated</i>	true
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	true
<i>SameNameSiblings</i>	false
PropertyDefinition	
<i>Name</i>	mcr:label
<i>RequiredType</i>	STRING
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	false
PropertyDefinition	
<i>Name</i>	mcr:parent
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	false
PropertyDefinition	
<i>Name</i>	mcr:child
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	true

Tabelle 6.1: Knotentypdefinition: mcrnt:object

Abbildung 6.4 verdeutlicht den durch diese Portierung erreichten Gewinn an zusätzlicher Semantik in dem JCR-Repository. Die Dokumente liegen nicht mehr „lose“ im Repository, sonder man kann „zusammen hängende“ erkennen. In der Visualisierung sind Vater-Sohn-Beziehungen oberhalb der Knoten angesetzt und dunkel. Die Umkehrrichtung (Sohn-Vater-Beziehung) ist unterhalb und in grau dargestellt. Diese Unterscheidung, natürlich nicht farbig, kann auch das Repository unternehmen, weil die Pfeile von unterschiedlichen, nicht eingezeichneten Eigenschaften ausgehen.

Diese Portierung erlaubt es uns, das Sicherstellen der referentiellen Integrität an das Repository auszulagern. Mögliche Fehlerquellen werden dadurch bereits reduziert, andere vielleicht geschaffen. Als Beispiel sei die Referenz auf die Kinder genannt (Eigenschaft `mcr:child`); es gibt keine Möglichkeit, den Typ der Kinder zu definieren. Umständlich bleibt auch weiterhin das Löschen. So müssen immer noch (vgl. Abschnitt 3.6.1) alle Kindobjekte „von Hand“ gelöscht werden, bevor ein Knoten gelöscht werden kann. Zusätzlich, aufgrund der referentiellen Integrität, können jetzt generell keine Knoten mehr gelöscht werden, auf die noch Referenzen gesetzt sind. Da sowohl Kinder ihren Vater referenzieren, wie auch umgekehrt, entsteht beim Löschen ein Paradoxon. Man kann den Vater nicht löschen, weil die Kinder auf ihn zeigen (über ihre Eigenschaft `mcr:parent`). Die Kinder können aber aufgrund der Eigenschaften `mcr:child` des Vaters ebenfalls nicht gelöscht werden. Das ist ein Punkt, an dem Transaktionen eine wichtige Rolle spielen können. So kann man verhindern, dass das System in einen inkonsistenten Zustand gerät. Das Löschen eines Knoten vom Typ `mcrnt:object` muss unter den eben besprochenen Umständen so aussehen:

1. Start Transaktion „Löschen“
2. Löschoperation:
 - (a) Für jedes Kind:
 - i. Setze Eigenschaft von Vaterknoten `mcr:child=null` (entfernt Referenz)
 - ii. Führe Löschoperation auf Kind aus
3. Ende Transaktion „Löschen“

Für den Fall, dass der Knoten jenseits der Vater-Sohn-Beziehung referenziert wird, müsste man gesondert untersuchen, wie in diesem Fall fortgefahren wird. Eine Lösung könnte ein Transaktionsabbruch sein, dann bleibt alles wie zuvor. Eine weitere Lösung besteht darin, die Referenzen auf den Knoten zu

entfernen, wo dies das Datenmodel (und damit der Knotentyp) zulässt. Neben dem Gewinn der referentiellen Integrität bedeutet diese Lösung der Portierung noch Nachteile im Löschverhalten von Knoten und in der Datenintegrität des Repositories. Das letzte Problem ist erst durch die Nutzung von JCR und dieser Portierungswahl entstanden, während das Löschproblem das aus 3.6.1 bekannte ist. Diese Probleme werden im folgendem Abschnitt angegangen.

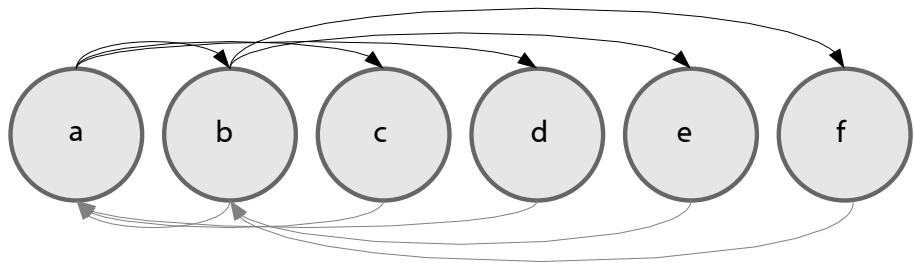


Abbildung 6.4: JCR-Sicht der Dokumenten-Hierarchie (Model 2)

6.2.3 Portierung mit Nutzung von Hierarchien

Bei der Portierung mit Nutzung der UUID haben wir die referentielle Integrität durch JCR sichergestellt, dennoch sieht das Löschverhalten noch so wie in 3.6.1 beschrieben aus. Das rekursive Löschen kann man noch zusätzlich in das JCR-Repository auslagern, indem man die MyCoRe-Hierarchie in eine JCR-Hierarchie abbildet. Das heißt, wenn ein MyCoRe-Objekt Kind eines anderen MCR-Objekts ist, dann muss der ihm entsprechende Knoten auch Kind des Knotens sein, der seinem Vater-MCR-Objekt entspricht. Dies lässt sich mit dem Knotentyp aus Tabelle 6.1 nicht erfüllen. Der nun in Tabelle 6.2 vorgestellte Knotentyp bietet diese Funktionalität.

Gewinn hierarchischer Unterstützung

mcrnt:hierarchyObject	
Name	Wert
NodeTypeNames	mcrnt:hierarchyObject
Supertypes	nt:base
IsMixin	false
HasOrderableChildNodes	false
PrimaryItemName	mcr:metadata
ChildNodeDefinition	
Name	mcr:metadata

Fortsetzung auf nächster Seite

Tabelle 6.2 – Fortsetzung von voriger Seite

Name	Wert
<i>RequiredPrimaryTypes</i>	[mcrnt:metadata]
<i>DefaultPrimaryType</i>	mcrnt:metadata
<i>AutoCreated</i>	true
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	true
<i>SameNameSiblings</i>	false
ChildNodeDefinition	
<i>Name</i>	mcr:child
<i>RequiredPrimaryTypes</i>	[mcrnt:hierarchyObject]
<i>DefaultPrimaryType</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	IGNORE
<i>Protected</i>	false
<i>SameNameSiblings</i>	true
PropertyDefinition	
<i>Name</i>	mcr:label
<i>RequiredType</i>	STRING
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	false

Tabelle 6.2: Knotentypdefinition: mcrnt:hierarchyObject

Die Knotentypdefinition aus Tabelle 6.2 benötigt keine Eigenschaften mehr für Kinder und Vater, wie noch in Tabelle 6.1. Diese Informationen gehen jetzt aus den Kinderknoten mit Namen `mcr:child` hervor. Diese Knoten sind wiederum hierarchische Knoten. Sollte ein Knoten versionierbar (vgl. Abschnitt 4.2.3) sein, so bedeutet das `IGNORE` bei `OnParentVersion`, dass eventuell versionierbare Kinderknoten keine neue Version bekommen. Dies entspricht dem Verhalten der Definition aus Tabelle 6.1 und bewirkt, dass

die Versions-History klein gehalten wird. Mit dem vorgestellten Knotentyp lässt sich das Löschproblem elegant an das Repository auslagern. Wie bei Abschnitt 6.2.2 jedoch, müssen „Seitenzeiger¹“ durch eigenen Code behandelt werden. Dieses jedoch lässt sich nicht auf das Repository verschieben, weil nur ein Löschverhalten definiert ist. So resultiert das Löschen eines Knotens, auf den noch Zeiger gerichtet sind, in eine `ReferentialIntegrityException` (Nüscheler u. a. 2005, Seite 188). Das verhindert zum Beispiel, dass ein Autor zu einem Buch gelöscht werden kann. Das Löschen eines Buches, wenn dieses auf einen Autor verweist, behindert es jedoch nicht. So wird in der Regel diese Ausnahme auch selten auftreten.

In **Abbildung 6.5** wird nun deutlich, dass die „JCR-Sicht“ der logischen Sicht entspricht, die in **Abbildung 6.2** bereits vorgestellt wurde. Damit wird es ermöglicht, dass Nicht-MyCoRe-Anwendungen mit größtmöglichen semantischen Informationen auf den Repository arbeiten können. Erst jetzt ist es für diese Anwendungen zum Beispiel möglich, einen logischen MyCoRe-Dokumententeilbaum zu exportieren, weil er dem logischen JCR-Knotenteilbaum entspricht.

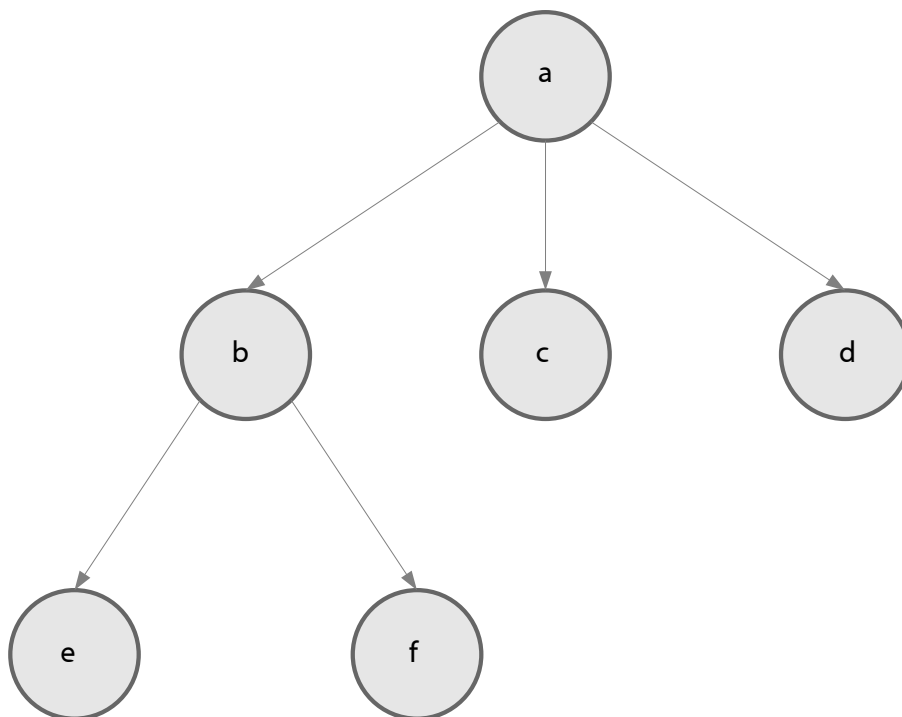


Abbildung 6.5: JCR-Sicht der Dokumenten-Hierarchie (Model 3)

¹Zeiger zu anderen Objekten jenseits der Vater-Sohn-Beziehung

6.2.4 Portierung der Metadatentypen von MyCoRe 1.0

In Abschnitt 3.2 wurde bereits einführend auf die Möglichkeiten von MyCoRe hingewiesen, unterschiedliche Datentypen für die Metadaten zu verwenden. Dem Anwendungsentwickler wird zusätzlich noch die Möglichkeit gegeben der vorhandenen Anzahl Datentypen (vgl. Anhang A), eigene hinzuzufügen. Diese Erweiterungsmöglichkeit muss natürlich bei einer Umstellung auf JCR noch erhalten bleiben. Einige Datentypen wie `MCRMetaBoolean` oder `MCRMetaLinkID` können ohne weitere Probleme auf die JCR-eigenen Datentypen `BOOLEAN` und `REFERENCE` überführt werden. Auch die komplexeren Datentypen stellen keine besonderen, sprich zusätzlichen, Anforderungen. Schon bei der XML-Abbildung (vgl. Abschnitt 3.2.2) mussten letztlich alle Datentypen in strukturierte Textdaten überführt werden. Den Datentyp `STRING` bietet auch JCR, so dass prinzipiell dieses Vorgehen auch bei JCR möglich ist. Darüber hinaus bietet JSR-170 noch eine Reihe weiterer elementarer Datentypen (vgl. Abschnitt 4.2.1), so dass die Modellierungsmöglichkeiten gesteigert werden können. Da die Integrität nicht mehr automatisch für jedes Metadatum von einer eigenen MyCoRe-JAVA-Klasse behandelt werden muss, bietet es sich an, für einen (komplexen) Metadatentyp, die „mixin“-Knotentypen von JCR zu verwenden. Folgendermaßen könnte die Definition dieses „mixin“-Typs aussehen:

mcrmix:checkable	
Name	Wert
<code>NodeTypename</code>	mcrmix:checkable
<code>Supertypes</code>	[]
<code>IsMixin</code>	true
<code>HasOrderableChildNodes</code>	true
<code>PrimaryItemName</code>	null
<code>PropertyDefinition</code>	
<code>Name</code>	mcr:verify
<code>RequiredType</code>	REFERER
<code>ValueConstraints</code>	[]
<code>DefaultValue</code>	null
<code>AutoCreated</code>	false
<code>Mandatory</code>	true
<code>OnParentVersion</code>	COPY
<code>Protected</code>	false
<code>Multiple</code>	false

Tabelle 6.3: Knotentypdefinition: `mcrmix:checkable`

Tabelle 6.3 zeigt die Knotentypdefinition von `mcrmix:checkable`. Knoten, die diesem „mixin“-Typ zugeordnet sind, müssen die Eigenschaft `mcr:verify` besitzen, die eine Referenz auf einen anderen Knoten ist. Bei dem referenzierten Knoten kann es sich um eine Konfigurationseinstellung handeln, wie dieser Knoten auf Gültigkeit zu überprüfen ist, also welche JAVA-Klasse seine (weitere) Integrität überprüft.

6.3 Architekturkonzepte

Als größtes Manko von MyCoRe in Kapitel 3 zeigte sich die enge Verquickung von Daten und Persistenzinformationen in der „Datenmodell- und XML-Schicht“, was so aus dem Architekturbild (vgl. Abbildung 3.1) nicht abzulesen war. Dort gab es eine „Persistenz-Schicht: Abbildung“, die für die Umsetzung von JAVA-Modell-Klassen verantwortlich sein sollte. Das diese Trennung nicht erfolgte, macht eine Anpassung von MyCoRe an andere Persistenzschnittstellen, wie z.B. JCR, sehr schwierig. So werden in diesem Abschnitt ein paar Möglichkeiten vorgestellt, wie man Daten in JAVA-Klassen von seiner Persistenz trennen kann.

Eine Möglichkeit, wie dies realisiert werden kann, führt der JCR-Standard vor. Es existieren eine Reihe von Schnittstellen, deren Implementierung in gewissen Grenzen frei ist. Angewandt auf MyCoRe würde dies bedeuten, dass die wichtigsten MyCoRe-Klassen wie `MCRObject` oder `MCRClassification` (vgl. Abschnitt 3.6) in Schnittstellen umgewandelt werden, deren Implementierung von der Persistenzschicht bereitgestellt wird. Um eine neue `MCRObject`-Instanz zu erzeugen, muss man dann eine *Factory*, die `MCRObjectFactory`, beauftragt werden. Das Design-Muster (*Design Pattern*) der Factory hilft, vom *Schnittstellen oder ...* Instanzierungsprozess eines Objekts zu abstrahieren. So können ohne grundlegende Kenntnisse der Persistenzschicht Objekte erzeugt werden, etwa mittels `MCRObjectFactory.createInstance()`, die wegen der implementierten Schnittstelle austauschbar und zugleich speziell angepasst sein können. Hier ist eine sorgfältige Planung nötig, damit alle nötigen Methoden für den Zugriff auf ein MyCoRe-Objekt integriert werden und zugleich das Zusammenspiel mit anderen MyCoRe-Komponenten möglich ist.

Dieses Vorgehen kommt selbstverständlich nur für ein paar ausgewählte Komponenten, wie die oben beschriebenen Klassen `MCRObject` und `MCRClassification` in Betracht. Schließlich will man Sachverhalte durch das Factory-Design-Pattern vereinfachen und nicht komplizierter machen. Dazu gehört auch das Überprüfen, wann das Muster effektiv genutzt werden kann. Ein Metadaten-Objekt kann zum Beispiel auf eine eigene Faktory verzichten,

wenn es nicht „losgelöst“ im System existieren kann und nur im Zusammenhang mit einem `MCRObjekt` vorkommt.

... ähnlicher Aufbau

Eine weitere Möglichkeit die Datenmodellschicht, die Business-Logik, von der Persistenz der Daten zu trennen, ist JAVA-Klassen in ihrem Aufbau mit ein paar grundlegenden Einschränkungen zu versehen.

1. **Parameterfreie Konstruktoren.** Es muss eine Instanz der Klasse erzeugt werden können, ohne das Parameter im Konstruktor erforderlich sind.
2. **Datenzugriff über Getter- und Setter-Methoden.** Die einzige Möglichkeit Daten in Objekten zu ändern oder zu lesen, ist über sogenannte Getter- und Setter-Methoden. Das bedeutet zum Beispiel, dass ein Feld mit Namen `fooBar` mittels `getfooBar()` gelesen und mit `setfooBar()` geschrieben wird.

Diese Anforderungen bewirken, dass unabhängig vom Aufbau und Funktionsumfang der Klasse, ihr „Zustand“ über die Getter-Methoden gesichert und mittels Setter-Methoden wieder hergestellt werden kann. JAVA bietet bereits Möglichkeiten sowohl diese Methoden bei beliebigen Objekten festzustellen, als auch Objekte von so aufgebauten Klassen zu sichern. Als Beispiel seien hier die seit JDK 1.4 vorhandenen Klassen `XMLDecoder` und `XMLEncoder` genannt, die auf diese Art JAVA-Objekte in XML-Code überführen und wieder herstellen können. Für eventuelle Validierungsschritte, die sonst im Konstruktor schon aufgerufen werden konnten, bedeutet dies, dass Überprüfungen bezüglich der Integrität eines Objektes auf den Zeitpunkt verschoben werden müssen, wenn zum ersten Mal Daten (mittels Getter-Methode) gelesen werden müssen. Zugehörig zu dieser Methode, kann auch eine andere Herangehensweise betrachtet werden. So kann man sich auf jeweils auf eine Getter- und Setter-Methode festlegen, die eine Liste von Objekten befüllt und ausliest, wobei jedes Objekt benannt ist. Das verringert die Zahl der Methoden, macht die Klassen „generischer“ und schränkt die Möglichkeiten dennoch nicht ein. Es erlaubt sogar „beliebig viele“ Daten in Objekte zu packen.

Beide vorgestellten Möglichkeiten bieten eine Trennung von Geschäftslogik und der Datenhaltung. Sie sind sogar kombinierbar. Jedoch bietet das Schnittstellen-Verfahren die Möglichkeit des „verzögerten Ladens“. Das heißt, es werden die Daten aus der Persistenz erst dann geladen, wenn sie wirklich benötigt werden. So kann das Volumen der zu übertragenden Daten verringert werden, der Speicherverbrauch minimiert und nicht benötigter Arbeitsaufwand verhindert werden. Stellt eine Klasse zum Beispiel eine Methode `delete()` bereit, so müssen für den Aufruf dieser Methode nicht alle Felder des Objektes voll besetzt sein. Gegenüber 3.6.1 ist dies ein großer Fortschritt.

Das Getter-Setter-Verfahren bietet den Vorteil, dass, wie bereits angedeutet, ein Objekt automatisch serialisiert werden kann, um es zum Beispiel über ein Netz von einem Rechnersystem zum nächsten zu verschicken. Dies kann nicht nur im Zusammenhang mit Web-Services wichtig sein.

6.4 Fazit

Ein Wechsel der bisherigen MyCoRe-Persistenzschicht hin zu JCR bietet eine Reihe von Vorteilen. Erstmals ist es möglich, dass externe Softwaresysteme den Funktionsumfang von MyCoRe erweitern können, ohne dass MyCoRe erweitert werden muss. Dies gelingt dadurch, dass MyCoRe einen Industriestandard für die Datenhaltung verwendet – für sämtliche Daten. Nur so können Komponenten von Fremdherstellern genutzt werden, die diese nicht extra für MyCoRe entwickelt haben. Die Sicht auf die Daten wird nicht von MyCoRe definiert und selbst interpretiert, sondern orientiert sich an JCR, was zukünftig eine breite Unterstützung durch die Industrie erhalten wird. So wird es möglich sein neben MyCoRe, Produkte anderer Hersteller zu betreiben, die auf gleiche Daten über eine gemeinsame, herstellerunabhängige Schnittstelle zugreifen. MyCoRe wollte eine herstellerunabhängige Repository-Schnittstelle bieten, aber ist MyCoRe nicht selbst ein Hersteller? PRO

Nachteilig ist jedoch, dass eine sinnvolle Verwendung von JCR zu einer Inkompatibilität mit MyCoRe 1.0 führt. Dieser „Haken“ wird aber dadurch gemindert, dass zweifellos nötige Architekturkorrekturen ebenfalls einen gewissen Grad an Inkompatibilität hervorrufen. Inkompatibilität ist nur dann ein Argument, wenn es eine unüberschaubare Anwenderbasis gibt und der Funktions- oder Leistungsgewinn als gering einzustufen ist. Letztere Punkte kann man negieren. Den Anwendungsentwicklern und Betreibern von MyCoRe-basierten Servern gilt es ein Migrationswerkzeug oder einen Migrationsleitfaden zu entwickeln, dann können sie von den Vorteilen der neuen Architektur mit geringem Aufwand profitieren. CONTRA

Wünschenswerte Weiterentwicklungen von JCR sind Regelungen, wie eigene Abfragesprachen in Repository integriert werden können. Auch der Wechsel von einem Repository zu einem eines anderen Herstellers wird durch mangelnde Definition von Zugriffskontrollen behindert. Unterstützen zwei Systeme Zugriffskontrollen via *Access Control Lists*, so wäre eine direkte Übertragung der Zugriffsrechtedefinitionen denkbar, aber zur Zeit nicht möglich. Als eine weitere wünschenswerte Erweiterung von JCR stellt sich die Unterstützung der Vererbung von Eigenschaften heraus. Dies wird zur Zeit in MyCoRe noch mit dem Umherkopieren von Werten erledigt und lässt sich durch ein Wechsel auf Potential bei JCR

JCR nicht überflüssig machen. Eine Standardisierung seitens JCR könnte dazu führen, dass auch in diesem Punkt der Codestand seitens MyCoRe verkleinert werden kann.

Trotz der noch bestehenden Probleme bietet JCR schon jetzt so gewichtige Vorteile, dass eine Nutzung dieses Standards von MyCoRe unbedingt in Betracht genommen werden sollte.

Kapitel 7

Zusammenfassung und Ausblick

„Ist die MyCoRe-Repository-Schnittstelle ein Filter?“ lautete die Frage, sich aus der Aufgabenstellung herleiten ließ. Nach Kapitel 3 konnte dies sehr eindeutig mit „Ja!“ beantwortet werden. Jedoch sind Schnittstellen grundsätzlich Filter. Sie filtern die Komplexität ihrer Implementierung weg. Wenn neben der Komplexität auch Funktionalität weggefiltert wird, ist entscheidend, für wen diese Funktionen wichtig sind. Das Beispiel MyCoRe hat gezeigt, dass der MyCoRe-Weg dazu führte, dass ein Großteil von vorhandenen Funktionen eines Backends nicht über die Schnittstelle(n) angeboten wird. Wo dies bislang erforderlich war, wurden über diesen Schnittstellen die Funktionen nachgebaut, um nicht in eine Abhängigkeitsfalle eines Herstellers zu gelangen. Aber ist die MyCoRe-Community nicht selbst ein Hersteller? Keine externen Programme können sinnvoll mit den Daten arbeiten, die MyCoRe getrennt über viele Komponenten verteilt, weil sie seine „Sicht“ (die MyCoRe-Sicht) nicht teilen (können). *Abhängig von MyCoRe?*

Als Hauptproblem bei MyCoRe stellt sich die fehlende Spezifikation heraus, aus der hervor geht, welche Komponenten bestimmte Funktionen bereitstellen und wie sie zusammen wirken. Die Folge ist eine verquickte Sammlung von Code, der mit der Zeit immer anfälliger für Fehler wurde und schlecht wartbar ist. Auch wenn dies nur ein Teil des gesamten MyCoRe-Codes betreffen mag, der in dieser Arbeit untersucht wurde, so muss es eine vordringliche Aufgabe sein, diese Architekturprobleme zu lösen.

Die Design-Phase von MyCoRe liegt schon lange zurück und MyCoRe ist, viel mehr als sein Name suggeriert, ein Content-Management-System; eher also ein MyCoMa. In der Zwischenzeit wurde geschaffen, was mit MyCoRe

Alternative: JCR

nicht geglückt ist: Ein wirkliche herstellerunabhängige Schnittstelle für JAVA, um auf Repositories zuzugreifen. Diese wurde in Kapitel 4 vorgestellt. Ein Vergleich beider Lösungskonzepte erfolgt auf Grundlage der Kapitel 3 und 4 über MyCoRe und JCR dann in Kapitel 5. Dabei stellte sich JCR als echte Alternative heraus, um Persistenzfunktionen in Anwendungen zu übernehmen.

Integration von JCR in MyCoRe

Im folgenden wurde geklärt, ob Content-Management-Systeme wie MyCoRe JCR als Schnittstelle für Content-Repositories nutzen können. Zu 100% ist dies nicht möglich, wie Kapitel 6 zeigte. Jedoch wurde auch gezeigt, dass ein Großteil der benötigten Funktionen durch JCR-Repositories abgedeckt werden kann. Dazu gibt es eine ganze Reihe zusätzlicher, bisher in MyCoRe nicht vorhandener, Funktionen „automatisch“ mit dazu. Auch die Anbindung von MyCoRe-fremder Software an das Repository profitiert von einer Integration von JCR in MyCoRe.

Offen blieben die Probleme, die in JCR 1.0 noch existieren. Die fehlende, von MyCoRe benötigte, Unterstützung wichtiger XPATH-Syntaxelemente und die fehlenden Exportmöglichkeiten für Zugriffsrechte machen einen Wechsel von einem Repository zu einem anderen zu einem schwierigerem Unterfangen.

Weitere Forschungsarbeiten

Am Beispiel von JCR zeigt sich auch ein grundlegendes Problem der Hersteller von Datenbank-Managementsystemen: Wie schaffe ich im Datenbankumfeld Strukturen, um Objekte effizient zu verwalten und zu durchsuchen. Bisherige Ansätze wie Suchbäume scheinen bei stark hierarchischen Strukturen nicht greifen zu können. Je mehr Hierarchiestufen kombiniert werden, desto mehr JOINS über Tabellen entstehen auf Seite des Datenbanksystems. Dieses grundlegende Problem anzugehen muss Ziel weiterer Forschungsarbeit sein.

Prototyp: MyCoRe mit JCR

Zu einem schwierigerem Unterfangen wird es auch werden einen Prototyp zu realisieren, der mittels JCR-Repsotitory-Schnittstelle läuft. Um ihn zu realisieren, muss in einem ersten Schritt die Architektur strikter aufgeteilt werden, was besonders die Trennung zwischen Geschäftslogik, also den Funktionen eines Systems, und Persistenzklassen, der Sicherung eines Systemzustandes, betrifft.

Nach dem ein lauffähiger Prototyp existiert, wird es einiges an Überzeugungsarbeit kosten, die MyCoRe-Community von der Richtigkeit dieses Schrittes zu überzeugen. Letzlich muss der Leidensdruck seitens der MyCoRe-Anwender nur groß genug sein, damit der Schritt akzeptiert wird. So lange „es doch läuft“ ist dies sicherlich nicht bei allen der Fall.

Vielleicht ist dann die Zeit reif, dass System jenseits der Bibliotheken einzusetzen und zu vertreiben. Es gibt da jemanden, der an einem guten Thomas-Aquinas-Archiv interessiert ist.

Anhang A

Metadatatypen aus MyCoRe 1.0

Metatype-Klasse	Beschreibung
MCRMetaAddress	Datentyp für Adressen mit folgenden Kinderelementen <ul style="list-style-type: none">• country. Land• state. Bundesland• zipcode. Postleitzahl• city. Stadt• street. Straße• number. Hausnummer
MCRMetaBoolean	Datentyp für <code>boolean</code> -Werte. Textknoten enthält Angabe für Wert: <ul style="list-style-type: none">• true. <code>true</code>, <code>yes</code>, <code>ja</code>, <code>wahr</code>• false. sonst

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von voriger Seite

Metatype-Klasse	Beschreibung
MCRMetaClassification	<p>Datentyp für Referenzen auf Klassifikationen. Folgende Attribute werden verwendet:</p> <ul style="list-style-type: none"> • classid. Bezeichner der Klassifikation • categid. Kategorie der Klassifikation die referenziert wird
MCRMetaDate	Datentyp für Datumangaben. Entweder im lokalen Format, oder im Format <i>yyyy – MM – dd</i> .
MCRMetaIFS	<p>Datentyp zur Positionsangabe von Dateien. Attributwerte:</p> <ul style="list-style-type: none"> • sourcepath. Pfad zum Speicherort der Dateien • maindoc. Hauptdokument des Derivats • ifsid. Wenn Derivat im IFS vorhanden ist, dann ist das seine ID (bekannt bei Derivat-Export)
MCRMetaInstitutionName	<p>Datentyp für Institutionsnamen. Kind-elemente</p> <ul style="list-style-type: none"> • fullname. Vollständiger Name der Institution • nickname. Kurzname der Institution • property. —
MCRMetaISBN	Datentyp für ISBN. Textknoten enthält gültige ISBN-Nummer.

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von voriger Seite

Metatype-Klasse	Beschreibung
MCRMetaLangText	Datentyp für einfachen Text. Optionales Attribut (form) kann Texttyp festlegen, Standardwert ist <code>plain</code> .
MCRMetaLink	<p>Datentyp für XLinks (DeRose u. a. 2001). Attributwerte</p> <ul style="list-style-type: none"> • xlink:type. entweder <code>locator</code> oder <code>arc</code> • xlink:href. Link-URI, bei <code>xlink:type='locator'</code> • xlink:label. Bezeichner, bei <code>xlink:type='locator'</code> • xlink:from. Link-Quelle, bei <code>xlink:type='arc'</code> • xlink:to. Linkziel, bei <code>xlink:type='arc'</code> • xlink:title. Titel
MCRMetaLinkID	<p>Wie MCRMetaLink, jedoch mit folgenden Einschränkungen.</p> <ul style="list-style-type: none"> • xlink:href. gültige MCRObjectID, bei <code>xlink:type='locator'</code> • xlink:from. gültige MCRObjectID, bei <code>xlink:type='arc'</code> • xlink:to. gültige MCRObjectID, bei <code>xlink:type='arc'</code>
MCRMetaNBN	Datentyp für NBN. Textknoten wird als NBN genommen.

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von voriger Seite

Metatype-Klasse	Beschreibung
MCRMetaNumber	<p>Datentyp für Nummernangaben (double) im Textknoten. Optional können folgende Attribute angegeben werden:</p> <ul style="list-style-type: none"> • measurement. Maßeinheit • dimension. Dimensionsangabe
MCRMetaPersonName	<p>Datentyp für Personennamen. Folgende Kindelemente sind möglich:</p> <ul style="list-style-type: none"> • firstname. Vorname • callname. Rufname • surname. Familienname • fullname. vollständiger Name • academic. akademische Titel • peerage. Adelstitel • prefix. Präfix
MCRMetaXML	<p>Datentyp für beliebiges XML. Attribute der Metadatenelements werden nicht übernommen, jedoch die der Kindelemente.</p>

Tabelle A.1: Vordefinierte Metadatatypen in MyCoRe Version 1.0

Anhang B

Knotentypen aus JCR 1.0

Dieser Anhang fasst noch einmal die besprochenen Knotentypen dieser Diplomarbeit zusammen. Ein komplette Übersicht aller eingebauten Knoten liefert (Nüscheler u. a. 2005, Kapitel 6.7).

B.1 Primäre Knotentypen

nt:unstructured	
Name	Wert
NodeTypeNames	nt:unstructured
Supertypes	nt:base
IsMixin	false
HasOrderableChildNodes	true
PrimaryItemName	null
ChildNodeDefinition	
<i>Name</i>	*
<i>RequiredPrimaryTypes</i>	[nt:base]
<i>DefaultPrimaryType</i>	[nt:unstructured]
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	VERSION
<i>Protected</i>	false
<i>SameNameSiblings</i>	true
PropertyDefinition	
<i>Name</i>	*

Fortsetzung auf nächster Seite

Tabelle B.1 – Fortsetzung von voriger Seite

Name	Wert
<i>RequiredType</i>	UNDEFINED
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	true
PropertyDefintion	
<i>Name</i>	*
<i>RequiredType</i>	UNDEFINED
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	COPY
<i>Protected</i>	false
<i>Multiple</i>	false

Tabelle B.1: Knotentypdefinition: nt:unstructured

B.2 „Mixin“ Knotentypen

mix:referenceable	
Name	Wert
NodeTypename	mix:referenceable
Supertypes	[]
IsMixin	true
HasOrderableChildNodes	false
PrimaryItemName	null
PropertyDefinition	
<i>Name</i>	jcr:uuid
<i>RequiredType</i>	STRING

Fortsetzung auf nächster Seite

Tabelle B.2 – Fortsetzung von voriger Seite

Name	Wert
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	true
<i>Mandatory</i>	true
<i>OnParentVersion</i>	INITIALIZE
<i>Protected</i>	true
<i>Multiple</i>	false

Tabelle B.2: Knotentypdefinition: `mix:referenceable`

mix:versionable	
Name	Wert
NodeTypename	mix:versionable
Supertypes	mix:referencable
IsMixin	true
HasOrderableChildNodes	false
PrimaryItemName	null
PropertyDefinition	
<i>Name</i>	jcr:versionHistory
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	["nt:versionHistory"]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	true
<i>Multiple</i>	false
PropertyDefintion	
<i>Name</i>	jcr:baseVersion
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	["nt:version"]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	true

Fortsetzung auf nächster Seite

Tabelle B.3 – Fortsetzung von voriger Seite

Name	Wert
<i>OnParentVersion</i>	IGNORE
<i>Protected</i>	true
<i>Multiple</i>	false
PropertyDefintion	
<i>Name</i>	jcr:isCheckedOut
<i>RequiredType</i>	BOOLEAN
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	[true]
<i>AutoCreated</i>	true
<i>Mandatory</i>	true
<i>OnParentVersion</i>	IGNORE
<i>Protected</i>	true
<i>Multiple</i>	false
PropertyDefintion	
<i>Name</i>	jcr:predecessors
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	[nt:version]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	true
<i>OnParentVersion</i>	COPY
<i>Protected</i>	true
<i>Multiple</i>	true
PropertyDefintion	
<i>Name</i>	jcr:mergeFailed
<i>RequiredType</i>	REFERENCE
<i>ValueConstraints</i>	[]
<i>DefaultValue</i>	null
<i>AutoCreated</i>	false
<i>Mandatory</i>	false
<i>OnParentVersion</i>	ABORT
<i>Protected</i>	true
<i>Multiple</i>	true

Tabelle B.3: Knotentypdefinition: mix:versionable

Literaturverzeichnis

AG Elektronisches Publizieren 2003

AG ELEKTRONISCHES PUBLIZIEREN: *DINI – Zertifikat Dokumenten- und Publikationsserver*, Deutsche Initiative für Netzwerkinformation (DINI), Empfehlung, November 2003. <http://www.dini.de/documents/Zertifikat.pdf>. – Elektronische Ressource

AIIM 2005

AIIM: *Glossary of ECM terms*. online. <http://www.aiim.org/article-aiim.asp?ID=27664>. Version: 2005

akademie.de 2005

AKADEMIE.DE: *Net-Lexikon*. Onlineenzyklopädie. <http://www.net-lexikon.de/>. Version: 2005

Bender 2005

BENDER, Thomas: *Benchmarking von ECM Systemen am Beispiel einer ePublishing Anwendung auf Basis von MyCoRe*, Albert Ludwigs Universität Freiburg, Diplomarbeit, April 2005. http://www.mycore.de/cvs/viewcvs.cgi/~checkout~/content/theses/document/docportal_derivate_07910405/Diplomarbeit-ThomasBender.pdf?rev=1.1. – Elektronische Ressource

Bray u. a. 1999

BRAY, Tim ; HOLLANDER, Dave ; LAYMAN, Andrew: *Namespaces in XML*. online. <http://www.w3.org/TR/REC-xml-names>. Version: 1999

Brodsky u. a. 1999

BRODSKY, Jay ; HUNT, Bruce ; KHOURY, Sami ; POPKIN, Laird ; ICE AUTHORIZING GROUP: *The Information and Content Exchange (ICE) Protocol Version 1.1*. online. <http://www.icestandard.org/Spec/SPEC-ICE1.1.htm>. Version: 1999

Cheung u. Matena 2002

CHEUNG, Susan ; MATENA, Vlada: *Java Transaction API (JTA); Version: 1.0.1B.* online. <http://java.sun.com/products/jta/>. Version: 2002

Clark u. DeRose 1999

CLARK, James ; DEROSE, Steve: *XML Path Language (XPath).* online. <http://www.w3.org/TR/xpath>. Version: 1999

Clemm u. a. 2003

CLEMM, Geoffrey ; CROSSLEY, Nick ; DOOLEY, John ; ELLISON, Tim ; RAYMOND, Peter ; SEDLAR, Eric: *WVCM: The Workspace Versioning and Configuration Management API.* online. <http://www.w3.org/TR/xlink/>. Version: 2003

Day Software 2004

DAY SOFTWARE: *JSR Public.* press release: online. http://www.day.com/content/en/company/media/press_releases/jsr170_spec_approved.main.html. Version: 2004

Day Software 2005a

DAY SOFTWARE: *Content Management JCP Specification Underway to Improve Interoperability.* press release: online. http://www.day.com/content/en/company/media/press_releases/jsr_170_aaim_ibm.main.html. Version: 2005

Day Software 2005b

DAY SOFTWARE: *Content Repository Standard Finalized.* press release: online. http://www.day.com/content/en/company/media/press_releases/jsr170_spec_approved.main.html. Version: 2005

DeRose u. a. 2001

DEROSE, Steve ; BROWN UNIVERSITY SCHOLARLY TECHNOLOGY GROUP ; MALER, Eve ; SUN MICROSYSTEMS ; ORCHARD, David ; JAMCRACKER: *XML Linking Language (XLink) Version 1.0.* online. <http://www.w3.org/TR/xlink/>. Version: 2001

Eco u. Coppock 1995

ECO, Umberto ; COPPOCK, Patrick: *A Conversation in Information.* (1995), Februar. http://carbon.cudenver.edu/~mryder/itc_data/eco/eco.html

Fielding 2005

FIELDING, Roy T.: *JSR 170 Overview – Standardizing the Content Repository Interface.* (2005). <http://www.day.com/content/en/>

product/jsr_170/white_paper.Par.0002.DownloadFile.0/
JSR_170_White_Paper.pdf

Gartner, Inc. 2004

GARTNER, INC.: The Cutting Edge: Global Content Management. (2004). <http://mediaproducts.gartner.com/gc/webletter/daysoftware/issue2/>

Goland u. a. 1999

GOLAND, Yaron Y. ; EMMET JAMES WHITEHEAD, Jr. ; FAIZI, Asad ; CARTER, Steve ; JENSEN, Del: *HTTP Extensions for Distributed Authoring – WEBDAV*. online. <http://www.ietf.org/rfc/rfc2518.txt>. Version: 1999

Häder u. Reuter 1983

HÄDER, Theo ; REUTER, Andreas: Principles of Transaction-Oriented Recovery. In: *ACM Computing Surveys* 15 (1983), Dec., Nr. 4, S. 287–317

Krebs 2004

KREBS, Kathleen: *Konzeption und Prototyp einer Client-/Server-Architektur für Online-Publishing basierend auf Webservicetechnologien*, Universität Hamburg; Regionales Rechenzentrum, Diplomarbeit, Juni 2004

Lagoze u. a. 2002

LAGOZE, Carl ; SOMPEL, Herbert V. ; OAI TECHNICAL COMMITTEE: *The Open Archives Initiative Protocol for Metadata Harvesting*. online. <http://www.openarchives.org/OAI/openarchivesprotocol.html>. Version: 2002

Lahnakoski 2005

LAHNAKOSKI, Kyle: Versioning. (2005), Februar. http://www.arcavia.com/kyle/Analysis/_Versioning.html

Lützenkirchen 2002

LÜTZENKIRCHEN, Frank: MyCoRe – Ein Open Source System zum Aufbau digitaler Bibliotheken. In: *Datenbank-Spektrum. Zeitschrift für Datenbanktechnologie* 2. Jahrgang (2002), November, Nr. 4, 23-27. http://miless.uni-essen.de/servlets/DerivateServlet/Derivate-11198/MyCoRe_DB-Spektrum.pdf. – ISSN 1618–2162

Lützenkirchen 2003

LÜTZENKIRCHEN, Frank: *MyCoRe und MILESS – Architektur und Technik*, IBM Forum: Von MILESS zu MyCoRe, Vortragsfolien, 2003. <http://miless.uni-essen.de/servlets/DerivateServlet/>

Derivate-11745/MyCoRe_Architektur_Technik_IBM_23.07.
2003.pdf. – Elektronische Ressource

Lützenkirchen u. a. 2005

LÜTZENKIRCHEN, Frank ; KUPFERSCHMIDT, Jens ; DEGENHARD, Detlev ;
BÜHLER, Johannes ; KRÖNERT, Ulrike ; STARKE, Ute ; ANDREAS, Trappe:
MyCoRe User Guide, MyCoRe Community, MyCoRe Begleitdokumentation,
2005. [http://www.mycore.de/cvs/viewcvs.cgi/~checkout~/mycore/
documentation/UserGuide/UserGuide.pdf?rev=1.18](http://www.mycore.de/cvs/viewcvs.cgi/~checkout~/mycore/documentation/UserGuide/UserGuide.pdf?rev=1.18). –
Elektronische Ressource

Lützenkirchen u. a. 2004

LÜTZENKIRCHEN, Frank ; KUPFERSCHMIDT, Jens ; DEGENHARDT,
Detlev ; BÜHLER, Johannes: *MyCoRe Programmer Guide*, MyCo-
Re Community, MyCoRe Begleitdokumentation, 2004. [http:
//www.mycore.de/cvs/viewcvs.cgi/~checkout~/mycore/
documentation/ProgGuide/ProgrammerGuide.pdf?rev=1.3](http://www.mycore.de/cvs/viewcvs.cgi/~checkout~/mycore/documentation/ProgGuide/ProgrammerGuide.pdf?rev=1.3). –
Elektronische Ressource

NISO 2001

NISO: *ANSI/NISO Z39.85 -2001 – Dublin Core Metadata Element Set*
[http://www.niso.org/standards/standard_detail.cfm?std_
id=725](http://www.niso.org/standards/standard_detail.cfm?std_id=725). – ISBN 1-880124-53-X

NISO 2003

NISO: *ANSI/NISO Z39.50 -2003 – Information Retrieval : Application Service
Definition & Protocol Specification* [http://www.niso.org/standards/
standard_detail.cfm?std_id=465](http://www.niso.org/standards/standard_detail.cfm?std_id=465). – ISBN 1-880124-55-6

Nüscheler u. a. 2005

NÜSCHELER, David ; PIEGAZE, Peeter ; JSR 170 EXPERT GROUP: *Content
Repository API for JAVA Technology Specification*. online. [http://www.jcp.
org/en/jsr/detail?id=170](http://www.jcp.org/en/jsr/detail?id=170). Version: 2005

Scheffler 2004

SCHEFFLER, Thomas: *Integration zusätzlicher Suchfunktionalität in MyCoRe
am Beispiel einer Volltextsuche*, Friedrich-Schiller-Universität Jena; Fakultät
für Mathematik und Informatik, Studienarbeit, 2004

Tate 2002

TATE, Bruce A.: *Bitter Java*. 1. Auflage. Manning Publication Co., 2002. –
ISBN 1-930110-43-X

Wikimedia Foundation 2005

WIKIMEDIA FOUNDATION: *Wikipedia, die freie Enzyklopädie*. Freie Onlineenzyklopädie. <http://de.wikipedia.org/>. Version: 2005

Notizen

