MyCoRe Progammer Guide

Frank Lützenkirchen
Jens Kupferschmidt
Detlev Degenhardt
Johannes Bühler
Heiko Helmbrecht

21. April 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Dieser Teil der Dokumentation ist vor allem für Applikationsprogrammierer gedacht. Er beschreibt die Designkriterien und ihre Umsetzung in der vorliegenden Version 1.0 . Mit Hilfe dieser Dokumentation sollte es Ihnen möglich sein, Details von MyCoRe zu verstehen und eigene Anwendungen konzipieren und implementieren zu können. Die Schrift wird stetig erweiteret.

1.Softwareentwicklung

1.1 Vorabbemerkungen

In diesem Kapitel soll kurz in die Design- und Entwicklungsmechanismen des MyCore-Projektes eingeführt werden.

1.2 CVS-Zugang

Der Zugang zum CVS-Server des MyCoRe Projekts für Entwickler erfolgt nach Freischaltung eines Accounts über SSH. Nach dem Freischalten sind folgende Umgebungsvariablen zu setzten:

```
CVS_RSH=ssh
CVSROOT=:ext:mcr_username@server.mycore.de:/cvs
export CVS_RSH CVSROOT
```

Es empfielt sich zuerst die MyCoRe Quellen herunterzuladen.

```
cvs -d:ext:mcr_username@server.mycore.de:/cvs checkout mycore
```

Danach können sie mit

```
cvs -d:ext:mcr_username@server.mycore.de:/cvs commit -m "Kommentar fürs CVS" file
```

Daten einstellen. Voraussetzung ist ein CVS-Login. Dieses können Sie bei Frank Lützenkirchen beantragen.[1] Soll in ein bestehendes Projekt eine neue Datei integriert werden, so legt man sie zunächst lokal im vorgesehenen (und bereits ausgecheckten!) Verzeichnis an. Dann merkt man sie mittels `cvs add<filename>` vor. Um sie dann global zuregistrieren, erfolgt ein (verkürzt): `cvs commit<filename>` . Neue Unterverzeichnisse werden auf die gleiche Weise angelegt. Weitere und sehr ausführliche Informationen gibt es zu Hauf im Internet[2].

1.3 Entwicklungsumgebungen

1.4 Tools

1luetzenkirchen@bibl.uni-essen.de

2http://www.cvshome.org/docs/

http://cvsbook.red-bean.com/translations/german/

http://panama.informatik.uni-freiburg.de/~oberdiek/documents/OpenSourceDevWithCVS.pdf

http://www.selflinux.org/selflinux/pdf/cvs_buch_kapitel_9.pdf

2.Allgemeines der Implementierungen

2.1 Klassen, Pakete und Verantwortlichkeiten

2.2 Allgemeine Klassen / Exception-Modell / MCRCache

2.3 Das API-Konzept allgemein

2.4 Die Session-Verwaltung

Mehrere verschiedene Benutzer und Benutzerinnen (oder allgemeiner Prinzipale) können gleichzeitig Sitzungen mit dem MyCoRe-Softwaresystem eröffnen. Während einer Sitzung werden in der Regel nicht nur eine sondern mehrere Anfragen bearbeitet. Es ist daher sinnvoll, kontextspezifische Informationen wie die UserID, die gewünschte Sprache usw. Für die Dauer der Sitzung mitzuführen. Da das MyCoRe-System mit mehreren gleichzeitigen Sitzungen konfrontiert werden kann, die zudem überverschiedene Zugangs wege etabliert sein können (z.B. Servlets, Kommandozeilenschnittstelle oder Webservices), muss das System einen allgemein verwendbaren Kontextwechsel ermöglichen.

Bei der Bearbeitung einer Anfrage oder Transaktion muss nicht jede einzelne Methode oder Klasse Kenntnis über die Kontextinformationen besitzen. Daher ist es sinnvoll, die Übergabe des Kontextes als Parameter von Methode zu Methode bzw. Von Klasse zu Klasse zu vermeiden. Eine Möglichkeit, dies zu bewerkstelligen ist der Einsatz von sog. Thread Singletons oder thread-local Variablen. Die Idee dabei ist, den Thread der den Request bearbeitet als Repräentation des Request selbstan zusehen. Dazu müssen die Kontextinformationen aller dings an den Thread angebunden werden, was seit Java1.2 mit Hilfe der Klassen java.lang.ThreadLocal bzw. java.lang.InheritableThreadLocal möglich ist. Jeder Thread hat dabei seine eigene unabhängig initialisierte Kopie der Variable. Die set() und get() Methoden der Klasse ThreadLocal setzen bzw. Geben die Variable zuruck, die zum gerade ausgeführten Thread gehört. Die Klassen der Sessionverwaltung von MyCoRe sind auf Basis dieser Technologie implementiert (siehe Abbildung).

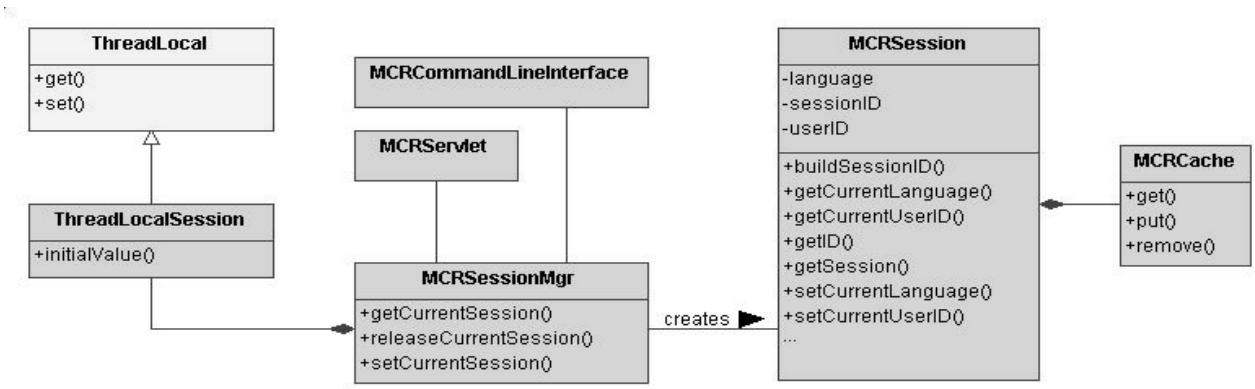


Abbildung 2.1: Die Klassen der Sessionverwaltung

Klienten der Sessionverwaltung sind alle Klassen, die Kontextinformationen lesen oder modifizieren wollen, wie zum Beispiel MCRServlet und MCRCommandLineInterface. Kontextinformationen werden als Instanzen der KlasseMCRSession abgelegt. Diese Klasse bietet Methoden zum Setzen und Lesen der Informationen, wie z. B. Der UserID der aktuellen Benutzerin, dergewünschten Sprache usw.

Die Klasse MCRSession besitzt einen statischen Cache, realisiert durch die Klasse MCRCache. Bei der Konstruktion einer Instanz von MCRSession wird zunächst über die Methode buildSessionID() eine eindeutige Iderzeugt und diese als Schlüssel zusammen mit dem Session-Objekt selbst im Cache abgelegt. Auf diese Weise hat man über die statische Methode getSession() Zugriff auf die zu einer bestimmten SessionID gehörende Instanz.

Damit die Instanzen von MCRSession als thread-local Variablen an den aktuellen Thread angebunden werden können, werden sie nicht direkt sondern über die statische Methode getCurrentSession() der Klasse MCRSessionMgr erzeugt und später gelesen. Beim ersten Aufruf von getCurrentSession() in einem Thread wird über die von java.lang.Threadlocal erbende, statische innere KlasseThreadLocalSession gewährleistet, dass eine eindeutige Instanz von MCRSession erzeugt und als thread-local Variable abgelegt wird. Der Zugriff auf die thread-local Variablen eines Threads kann nur über die Klasse ThreadLocal (bzw. InheritableThreadLocal) erfolgen. Auf diese Weise ist sichergestellt, dass bei nachfolgenden Aufrufen von getCurrentSession() genau die zum aktuellen Thread gehörende Referenz auf die Instanz vonMCR Session zuruckgegeben wird.

Mit der statischen Methode MCRSessionMgr.setCurrentSession() ist es möglich, ein bereits vorhandenes Session-Objekt explizit als thread-local Variable an den aktuellen Thread zu binden. Dies ist z.B. In einer Servlet-Umgebung notwendig, wenn die Kontextinformationen in einem Session-Objekt über eine http-Session mitgeführtwerden. (Aktuelle Servlet-Engines verwenden in der Regel zwar Thread-Pools für die Bearbeitung der Requests, aber es ist in keiner Weise sichergestellt, dass aufeinanderfolgende Requests mit dem selben Kontext wieder den selben Thread zugewiesen bekommen. Daher muss der Kontext in einer httpSession gespeichert werden.)

Die Methode MCRSessionMgr.releaseCurrentSession() sorgt dafür, dass das thread-local Session-Objekt eines Threads durch ein neues, leeres Objekt ersetzt wird. Dies ist in Thread-Pool-Umgebungen wichtig, weil es sonst moglich bzw. sogar wahrscheinlich ist, dass Kontextinformationen an einem Thread angebunden sind, dieser Thread aber bei seiner Wiederverwendung in einem ganz anderen Kontextarbeitet. Code-Beispiele zur Verwendung der Session-Verwaltung finden sich in org.mycore.frontend.servlets.MCRServlet.doGetPost().

2.5 Das Vererbungsmodell

In MyCoRe ist innerhalb des Datenmodells für die Metadaten die Möglichkeit einer Vererbung vorgesehen. Dies ist fest in den Kern implementiert und wird ausschließlich durch die steuernden Metadaten des jeweiligen Datensatzes festgelegt. Das heißt, für eine Datenmodell-Definition (z. B. document) können Datensätze mit (Buch mit Kapiteln) und ohne (Dokument) nebeneinander eingestellt werden. Wichtig ist nur, dass die Vererbung nur innerhalb eines Datenmodells oder eines, welches die gleiche Struktur aufweist, jedoch andere Pflichtfelder hat, funktioniert. Vererbung zwischen Datenmodellen mit verschieden Metadaten ist ausgeschlossen.

Im folgenden soll die Vererbung anhand des XML-Syntax zweier Metadaten-Objekte verdeutlicht werden. Beim Laden der Daten wird dann eine Eltern-Kind-Beziehung im System aufgebaut und abgespeichert.

```
<mycoreobject ... >
...
<metadata xml:lang=“de“>
<titles class=“MCRMetaLangText“ heritable=“true“ notinherit="false"..>
<title xml:lang=“de“>Buchtitel</title>
</titles>
<authors class=“MCRMetaLangText“ heritable=“true“ notinherit="false"..>
<author xml:lang=“de“>Erwin der Angler</author>
</authors>
<sizes class=“MCRMetaLangText“ heritable=“false“ notinherit="false"..>
<size xml:lang=“de“>100 Seiten</size>
</sizes>
...
</metadata>
....
</mycoreobject>
```

Abbildung 2.2: Auszug aus dem Metadaten-Objektes des Elternsatzes

```
<mycoreobject ... >
…
<metadata xml:lang=“de“>
<titles class=“MCRMetaLangText“ heritable=“true“ notinherit="false"..>
<title xml:lang=“de“>Kapitel 1</title>
</titles>
<sizes class=“MCRMetaLangText“ heritable=“true“ notinherit="true"..>
<size xml:lang=“de“>14 Seiten</size>
</sizes>
…
</metadata>
….
</mycoreobject>
```

Abbildung 2.3: Auszug aus dem Metadaten-Objektes des Kindsatzes

Die Beiden Datensätze sollen folgendes Szenario widerspiegeln:

- Der Titel soll für die Kind-Daten übernommen werden und durch diese um die Kapitelüberschriften ergänzt werden.
- Die Autorendaten sind an alle Kinder zu vererben.
- Der Umfang des Werkes ist je nach Stufe anzugeben, also für das Gesamte Buch die Gesamtzahl der Seiten und für ein Kapitel die Anzahl dessen Seiten.

Entscheidend für die Umsetzung sind folgende Dinge:

- Das Attribut **heritable** sagt, ob ein Metadatum vererbbar (**true**) oder nicht (**false**) sein soll.
- Das Attribut **notinherit** sagt, ob das Matadatum von dem Elterndatensatz nicht (**true**) geerbt werden soll. Andernfalls wird geerbt (**false**).
- Es muss erst der Elterndatensatz eingespielt werden. Anschließend können die Kindsätze der nächsten Vererbungsebene eingespielt werden. Enkelsätze folgen darauf, usw. Bitte beachten Sie das unbedingt beim Restore von Sicherungen in das System. MyCoRe ergänzt intern die Daten der Kind-Datensätze für die Suchanfragen um die geerbten Daten.
- Das Attribut **inherited** ist per Default auf **0** gesetzt und beschreibt die Anzahl der geerbten Daten. Das Attribut wird von System automatisch gesetzt. Kinder erhalten, wenn sie Daten geerbt haben, inherited=1 usw., je nach Stufe.

Die Ausgabe des Eltern-Datensatzes nach einer Query entspricht dem der Dateneingabe, lediglich im XML-structure-Teil wurde ein Verweis auf die Kinddaten eingetragen.[3]

Das unten stehende Listing zeigt den Kind-Datensatz, wie er von System nach einer erfolgreichen Anfrage im Resultats-Container zurückgegeben wird. Dabei ist deutlich die Funktionalität der MyCoRe-Vererbungsmechanismen zu erkennen.

3 Siehe XML-Syntax im User Guide.

```
<mycoreobject ... >
…
<metadata xml:lang=“de“>
<titles class=“MCRMetaLangText“ heritable=“true“ notinherit="false"..>
<title inherited="1" xml:lang=“de“>Buchtitel</title>
<title inherited="0" xml:lang=“de“>Kapitel 1</title>
</titles>
<authors class=“MCRMetaLangText“ heritable=“true“ notinherit="false"..>
<author inherited="1" xml:lang=“de“>Erwin der Angler</author>
</authors>
<sizes class=“MCRMetaLangText“ heritable=“false“ notinherit="false"..>
<size inherited="0" xml:lang=“de“>14 Seiten</size>
</sizes>
…
</metadata>
….
</mycoreobject>
```

Abbildung 2.4: XML-Syntax eines Kind-Datensatzes als Query-Resultat

3.Funktionsprinzipien und Implementierungen von Kernkomponenten

3.1 Das QueryModell von MyCoRe

MyCoRe bemüht sich, das Syntax-Modell der Suchabfragen an die existierenden Standards des W3C für XML Path Language (XPath) Version 1.0 (W3C Recommendation 16. November 1999) anzugleichen, da die derzeit häufigste Abfrage-Syntax im XML-Bereich ist. Dabei muss aber auch Rücksicht auf die Spezifik des MyCoRe-Systems zur Suche von Objekten und Metadaten genommen werden. Daher ist MyCoRe nicht 100% XPath/XQuery -konform, es wird sich jedoch bemüht, bestmöglich die Spezifkationen einzuhalten. Grund für diese Abweichungen sind die Praxisorientierung des MyCoRe-Systems, vor allem im Bereich digitaler Bibliotheken. So sollen verschiedene Persistence-Systeme zum Einsatz kommen, welche sehr unterschiedliche Abfrage-Syntax, auch außerhalb der XML-Welt, implementieren. Daher stellt der in diesem Abschnitt beschriebene Syntax nur einen recht keinen Teil der Möglichkeiten der W3C Spezifikationen dar. Er genügt aber in der Praxis, um recht komplexe Projekte zu realisieren. Eine Annäherung an Xpath2.0 und Xquery 1.0 wird erster folgen, wenn die Standardisierungsphase im wesentlichen abgeschlossen ist.

MyCoRe muss neben der eigentlichen Query noch erfahren, welche Datenmodell-Typen abzufragen sind. Auch dafür ist die Ursache in der Persistence-Unabhängigkeit von MyCoRe zu suchen. Es ist notwendig, die Typen auf die entsprechenden Stores zu mappen, damit die Suche erfolgreich ist. Als Beispiel soll hier der Content Manager 8 ItemType oder die Umsetzung unter eXist stehen. Möglich ist sowohl einzelne Typen, wie auch eine Liste davon anzugeben. Die Liste ist ein String mit folgendem Syntax:

```
Stringtype_list="type1[,...]"
```

Wichtig ist nur, dass die Elemente, nach denen gefragt wird, in allen Datenmodellen der Typen vorkommen (sonst könnte das Ergebnis der Suche eine leere Resultatsliste sein). Normalerweise wird die Type-Liste in der Applikation via Konfiguration festgelegt[4] und an die entsprechenden Query-Schnittstellen im API (MCRQueryBase) übergeben, z. B. durch das SearchMaskServlet.

3.1.1 Operatoren

MyCoRe verwendet nur die folgenden Operatoren innerhalb eines Test. Dies begründet sich mit der Kompatibilität und Umsetzung gegenüber den einzelnen Persitence-Layern. Wenn weitere Operatoren benötigt werden, so müssen diese in **ALLEN** Persitence-Implementierungen eingebaut werden.

```
SingleQuery        :: OutPath[ Tests ]
                      Tests
Tests              :: Test{ AND | OR Test ... }
Test               :: InPath Operator Value
```

- Operator = - ist für alle Values zulässig

4 mycore.properties: MCR.type_alldocs=document

- Operator != - ist für alle Values zulässig
- Operator < - ist nur für Datums- und Zahlenangaben zulässig
- Operator <= - ist nur für Datums- und Zahlenangaben zulässig
- Operator > - ist nur für Datums- und Zahlenangaben zulässig
- Operator >= - ist nur für Datums- und Zahlenangaben zulässig
- Operator like – versucht im Value den angegebenen String zu finden, als Wildcard ist * zulässig
- Operator contains – arbeitet wie like, ist eien TextSearch-Engine verfügbar, so erfolgt die linguistische Suche darin

Die Operatoren **like** und **contains** sind eine Ergänzung von MyCoRe um mit Textsuch-Mechanismen arbeiten zu können. Sie sind in der von MyCoRe benötigten Syntax-Form so nicht Bestandteil der W3C Spezifikationen, haben sich aber in der Praxis bewährt.

3.1.2 Pfadangaben

Die Pfadangaben für eine Single Query können einfach oder einmal geschachtelt sein, jenachdem, wie die einzelnen Text- bzw. Attributknoten des XML-Datenmodells logisch zusammengehören. Alle hier aufgeführten Möglichkeiten sind relativ zu XML-Dokument-Wurzeln. Andere Pfadangaben wie beispielsweise attribute:: für @ sind aus Gründen der Kompatibilität zu den einzelnen Persitence-Layern nicht erlaubt (und in der Praxis auch nicht erforderlich).

```
OutPath   :: a
             a/b
InPath    :: SpecialNodeFuncition
             @d
             text()
             c
             c/text()
             c/@d
             c/d/@e
             c/d/text()
SpecialNodeFuncition::      ts()
                            text()
                            doctext()
```

Hier noch einige Hinweise:

- ***** als SpecialNodeFuncition darf allein nur in einer Single Query ohne OutPath angegeben werden. Es erfolgt dann die Suche über alle für TextSearch markierte Metadaten.
- **text()** als SpecialNodeFuncition darf allein nur in einer Single Query ohne OutPath angegeben werden. Es wird nach 'ts()' überführt.
- **doctext()** als SpecialNodeFuncition ist für die Abfrage des TextSearch des Dokument-Objektes vorgesehen.
- In MyCoRe kann bei Bedarf der der Applikation um weitere SpecialNodeFuncition ergänzt werden.

Nun einige gültige Beispiele:

```
text()contains"..."
*like"..."
@ID="..."
metadata/rights/right="..."
metadata/rights/right/text()="..."
metadata/masse/mass[text()="..."and@type="..."]
doctext()contains"..."
```

3.1.3 Abfragen von Objekt-Metadaten

Unter Objekt-Metadaten sind alle Datenmodell-Typen zu verstehen, welche NICHT **class** oder **derivate** sind[5]. Alle XML-Files der Objekt-Metadaten Typen haben als Master-Tag /mycoreobject und genau auf diesen Knoten als Return-Value zielen auch alle Queries unter MyCoRe. Der allgemeine Syntax ist also:

```
Query::            OneQuery{ AND | OR OneQuery { AND | OR ... }}
OneQuery::         /mycoreobject[SpecialNodeFuncition]
```

Dies bedeutet, es können mehrere Abfragen hintereinander mit AND oder OR verknüpft werden. Für jedes Metadatum des Datenmodells ist eine OneQuery zu formulieren. Das diese Anfragen alle auf denselben Datenraum laufen,dafür sorgt die oben beschriebene Festlegung des Type-Elementes. Somit erhalten Sie ein korrektes Gesamtergebnis.

3.1.4 Das Resultat der Query

Alle Antworten als Resultat der Anfrage werden in einem XML-Conatiner zusammengefasst und der Anwendung zurückgegeben. Der Aufbau des Containers ist dabei Persitence-unabhängig. Nachfolgend die XML-Struktur des Resultates einer Query:

```
<mcrresults parsedByLayoutServlet="...">
<mcrresult host="..." id="..." rank="0" hasPred="..." hasSucc="...">
<mycoreobject ...>
</mycoreobject>
</mcrresult>
...
</mcrresults>
```

Abbildung 3.1: XML Syntax des Query-Resultates

- Das Attribut **parsedByLayoutServlet** ist ein Flag, welches eine etwaige Vorverarbeitung des Ergebnisses durch das LayoutServlet anzeigt.
- Das Attribut **host** beinhaltet entwerden '**local**' oder den Hostalias des Servers, von dem das Resultat stammt.

5 Die Typen class und derivate sind reservierte Typen.

- Das Attribut **id** ist die entsprechende MCRObjectID des Resultates.
- Das Attribut **hasPred** gibt mit **true** an, dass dieses Resultat einen Vorgänger hat. Dies ist für die Präsentation der Einzeldaten von Interesse.
- Das Attribut **hasSucc** gibt mit **true** an, dass dieses Resultat einen Nachfolger hat. Dies ist für die Präsentation der Einzeldaten von Interesse.

3.1.5 Abfragen von Derivaten

Für die Derivate gelten die selben Regeln für die Queries. Bachten Sie jedoch, dass das Datenmodel fest vorgeschieben ist und das Master-Tag anders heisst. Das Resultat auf eine Anfrage wird auch in einen **mcrresults**-Container verpackt.

```
Query     :: OneQuery{ AND |OR OneQuery { AND | OR ... }}
OneQuery  :: /mycorederivate[SpecialNodeFuncition]
```

3.1.6 Abfragen von Klassifkationen

Klassifkationen werden in MyCoRe-Anwendungen erfahrungsgemäß am häufigsten abgefragt. DieKlassifkationen können nur nach einem sehr festen Schema abgefragt werden, wobei entwerder die gesamte Klassifkation oder nuer eine einzelne Kategorie zurückgegeben wird. DieXML-Struktur entspricht dabei immer der einer Klassifkation.

```
Query::   /mycoreclass[@ID="..." { and @category like "..."}]
```

3.2 Die Benutzerverwaltung

Dieser Teil der Dokumentation beschreibt Funktionalität, Design, Implementierung und Nutzung des MyCoRe Subsystems für die Benutzerverwaltung.

3.2.1 Die Geschäftsprozesse der MyCoRe Benutzerverwaltung

Das Benutzermanagement ist die Komponente von MyCoRe, in der die Verwaltung derjenigen Personen geregelt wird, die mit dem System umgehen (zum Beispiel als Autoren Dokumente einstellen). Zu dieser Verwaltung gehört auch die Organisation von Benutzern in Gruppen. Eine weitere Aufgabe dieser Komponente ist das Ermöglichen einer Anmelde-/Abmeldeprozedur.

Ein Use-Case Diagramm (sieheAbbildung) soll eine Reihe typischer Geschäftsprozesse des Systems zeigen (ohne dabei den Anspruch zu haben, alle Akteure zu benennen oder alle Assoziationen der Akteure mit den Geschäftsprozessen zu defnieren).

Offensichtlich dürfen nicht alle Akteure des Systems die Berechtigung haben, alle Geschäftsprozesse durchführer zu können. Daher musse in System von Privilegien und Regeln implementiert werden: Benutzer/innen haben Privilegien (z.B. die Berechtigung, neue Benutzer/innen zu erstellen) Die Vergabe der Privilegien wird durch die Mitgliedschaft der Benutzer/innen in Gruppen geregelt. Darüber hinaus muss das System definierten Regel gehorchen.

So genügt z.B. Das Privileg '**add user to group**' allein nicht, um genau das Hinzufügen eines benutzers zu einer Gruppe definieren zu können. Die Regel ist in diesem Fall, jeder andere Benutzer mit diesem Privileg aber maximal Zugehörigkeit zu Gruppen vergeben kann, in denen er oder sie selbst Mitglied ist. Auf diese Weise wird verhindert, dass sich ein Benutzer oder eine Benutzerin selbst höhere Privilegien zuweisen kann. Die Privilegien und Regeln der MyCoRe-Benutzerverwaltung werden weiter unten ausgeführt.

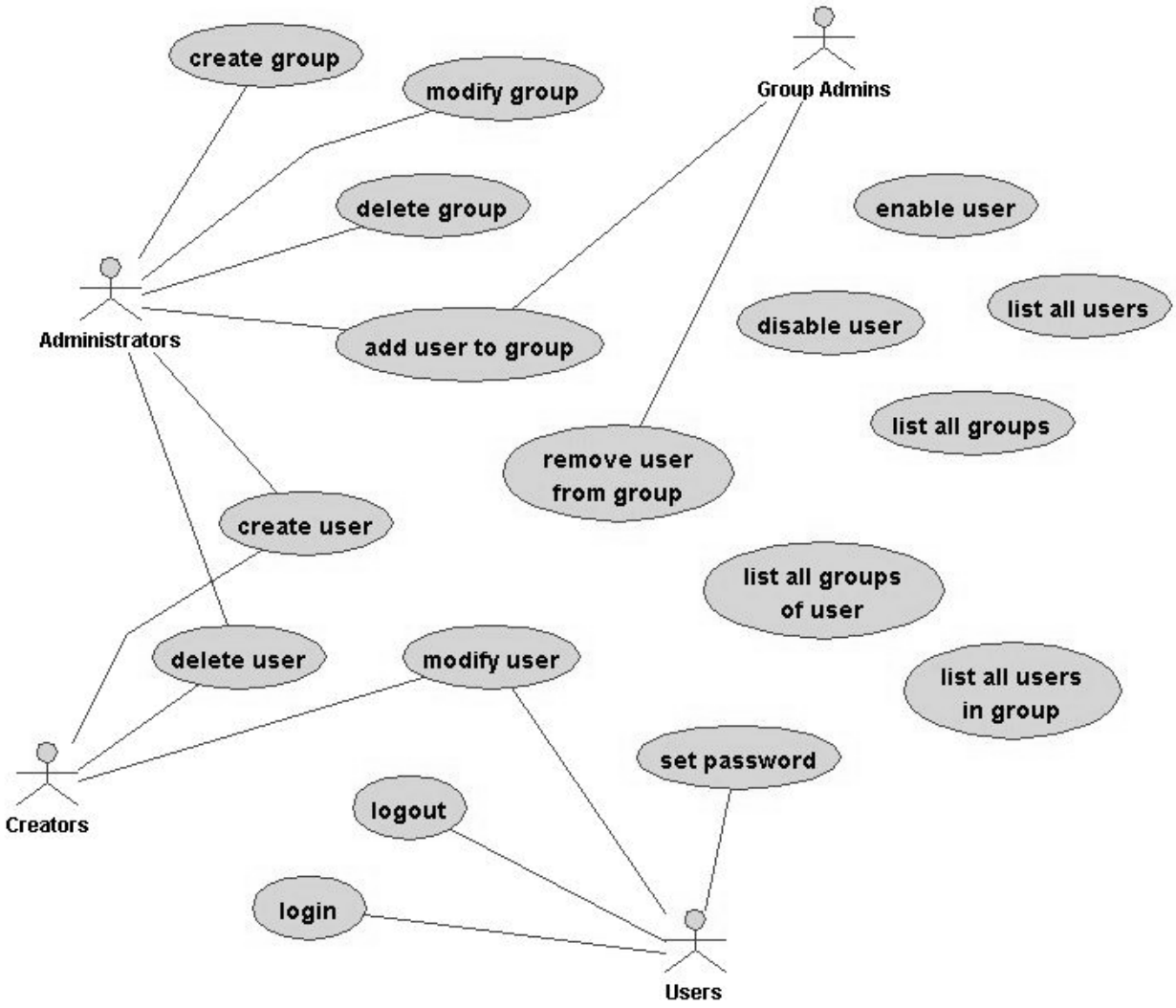


Abbildung 3.2: Geschäftsprozesse der Benutzerverwaltung in MyCoRe

3.2.2 Benutzer, Gruppen, Privilegien und Regeln

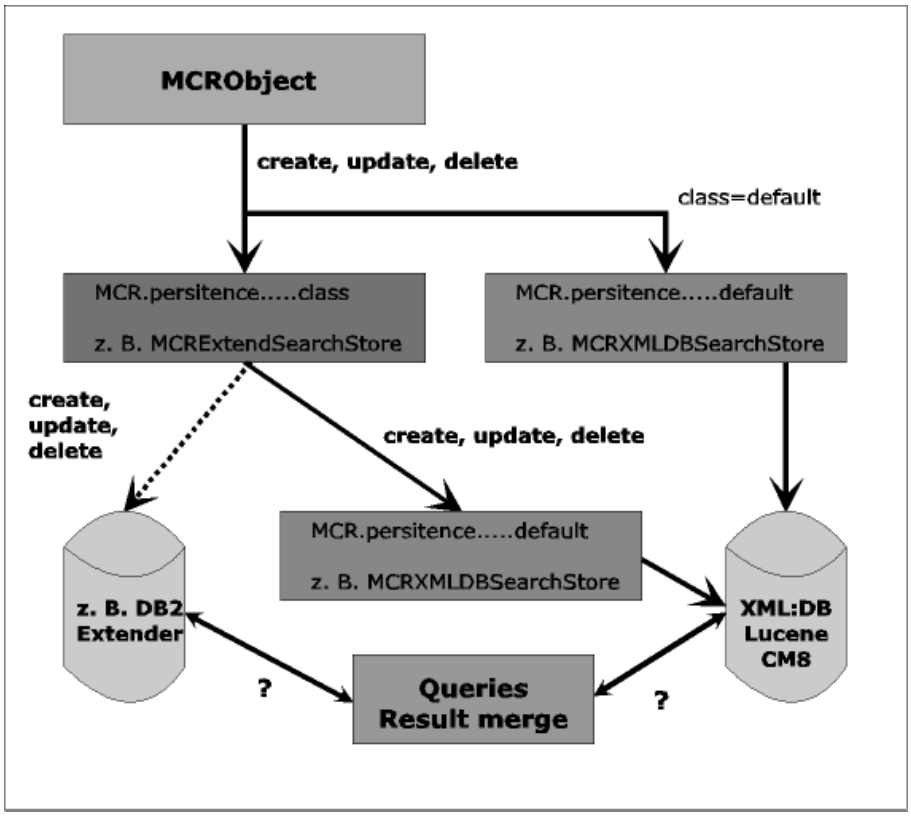
Die Attribute von Benutzern/innen des Systems können in drei Bereiche klassifiziert werden, den account-Informationen wie ID, Passwort, Beschreibung usw., den Adress-Informationen wie Name, Anrede, Fakultätszugehörigkeit usw. sowie den Informationen über die Mitgliedschaft zu Gruppen. Die aktuell implementierten Benutzerattribute kann man an folgender beispielhafter XML-Darstellung erkennen: **to be continued...**

3.3 Der Backend-Store

Im Backend-Bereich muss zwischen den Stores für die Metadaten und denen für die eigentlichen Objekte unterschieden werden. Während erstere recht umfangreiche Suchen auf einzelne Datenfelder gestatten müssen, dienen letztere meist nur der Datenspeicherung. Lediglich in einigen Fällen wird diese mit einer Volltextsuche ergänzt.

3.3.1 Backend-Stores für die Metadaten allgemein

Momentan sind im MyCoRe-Kern zwei Backends für die Speicherung der Metadaten implementiert. Das System ist so gestaltet, dass eine Ergänzung dieser Stores durch weitere applikationsabhängige Datenbasen einfach möglich ist. Hierzu kann entwerder der Standard-Store entsprechend erweitert werden oder die im Applikationsbereich vorhandene Klasse MCRExtendStore wird entsprechend ausgebaut, was der bessere Weg ist. Welche Store-Klassen Verwendung finden, wird in der Konfiguration festgelegt, für XML:DB in *mycore.properties.xmldb* und für CM8 in *mycore.properties.cm8*. Per Standardeinstellung sind die Klassen für den Store und den Default gleich, so dass der Extender nicht angesprochen wird. Den ganzen Zusammenhang soll das folgende Schema verdeutlichen.



3.3.2 Das Metadaten-Backend für IBM Content Manager 8.2

Hinweis:

Strings, welche NUR Zahlen enthalten, werden als Zahl interpretiert. Läuft die Anfrage gegen ein als **VarChar** definiertes Attribut, kommt es zum Fehler durch den CM. Ergänzen Sie den Zahlen-

String in der Suche z.B. mit einem ***** und benutzen Sie den Like-Operator. Beispiel **like 0004***.

3.3.3 Das Metadaten-Backend für XML:DB

Hinweis:

eXist ignoriert derzeit einige Schachtelungskonstrukte eines Test mit '[...]'. Daher kann das Ergebnis der Anfrage mehr Treffer haben, als bei korrekter Ausführung der Query.

3.4 Die Frontend Komponenten

3.4.1 Erweiterung des Commandline Tools

Dieser Abschnitt wird sich mit der Struktur des Commandline-Tools und dessen Erweiterung mit eigenen Kommandos beschäftigen. Dem Leser sei vorab empfohlen, den entsprechenden Abschnitt im MyCoRe-UserGuide durchzuarbeiten.

Das Commandline-Tool ist die Schnittstelle für eine interaktive Arbeit mit dem MyCoRe-System auf Kommandozeilen-Basis. Sie können dieses System ebenfalls dazu verwenden, mittels Script-Jobs ganze Arbeitsabläufe zu automatisieren. Dies ist besonders bei der Massendatenverarbeitung sehr hilfreich. Im MyCoReSample werden Ihnen schon in den Verzeichnissen *unixtools* bzw. *dostools* eine ganze Reihe von hilfreichen Scripts für Unix bzw. MS Windows mitgegeben.

All diese Scripts basieren auf dem Shell-Script *bin/mycore.sh* bzw. *bin/mycore.cmd*, welches im Initialisierungsprozess der Anwendung via **ant** mit gebaut wird (ant create.unixtools bzw. ant. create.dostools). sollten Sie zu einem späteren Zeitpunkt eventuell einmal **.jar*-Dateien in den *lib*-Verzeichnissen ausgetauscht haben oder sonstige Änderungen hinsichtlich des Java-CLASSPATH durchgeführt haben, so führen Sie für ein Rebuild des MyCoRe-Kommandos ein ant scripts durch.

Die nachfolgende Skizze soll einen Überblick über die Zusammenhänge der einzelnen Java-Klassen im Zusammenhang mit der nutzerseitigen Erweiterung des Commandline-Tools geben.

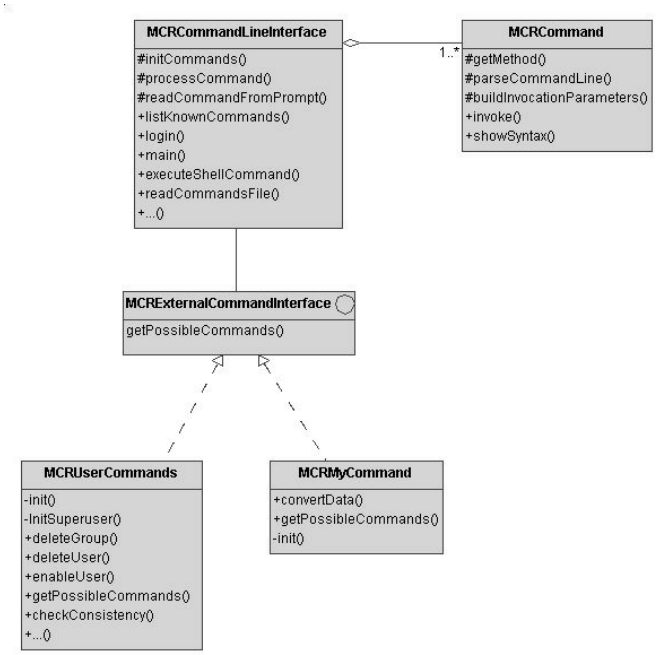


Abbildung 3.3: Zusammenhang der Java-Klassen

Es ist relativ einfach, weitere Kommandos hinzuzufügen. Im MyCoReSample sind bereits alles nötigen Muster vorhanden.

1. Im Verzeichnis *~/docportal/sources/org/mycore/frontend/cli* finden Sie eine Java-Klasse MCRMyCommand.java. Diese ist ein Muster, kopieren Sie sie in eine Java-Klssse z. B. MCRTestCommand.java . Die Klasse **muss** im Package **org.mycore.frontend.cli** liegen.
2. Ersetzen Sie alle MCRMyCommand-String durch MCRTestCommand .
3. Im Constructor werden nun alle neuen Kommandos definiert. Hierzu werden der ArrayList **command** jeweils zwei weitere Zeilen hinzugefügt. Die erste enthält den Text-String für das Kommando. Stellen wo Parameter eingefügt werden sollen sind mit {...} zu markieren, wobei ... eine fortlaufende Nummer beginnend mit 0 ist.[6] In der Zweiten Zeile ist nun der Methodenaufruf anzugeben. Für jeden Parameter ist das Schlüsselwort **String** anzugeben.
4. Nun muss das eigentliche Kommando als Methode dieser Kommando-Klasse implementiert werden. Orientieren Sie sich dabei am mitgelieferten Beispiel.[7]
5. Compilieren Sie nun die neue Klasse mit cd ~/docportal; ant jar
6. Als letztes müssen Sie die Klasse in das System einbinden. Die mit dem MyCoRe-Kern mitgelieferten Kommandos sind bis auf die Basis-Kommandos über die Property-Variable **MCR.internal_command_classes** in *~/docportal/mycore.properties* dem System bekannt gemacht. Für externe Kommandos steht hierfür im Konfigurations-File *mycore.properties.application* die Variable **MCR.external_command_classes** zur Verfügung. Hier können Sie eine mit Komma getrennte Liste Ihrer eigenen Kommando-Java-Klassen angeben.
7. Wenn Sie nun mycore.sh bzw. mycore.cmd starten und DEBUG für den Logger eingeschaltet haben, so sehen Sie Ihre neu integrierten Kommandos.

6 siehe Beispielcode
7 Die Methode convertData sollten Sie in Ihrer Klasse löschen. Ebenso die Definition in der commands-ArrayList.

3.4.2 Das Zusammenspiel der Servlets mit dem MCRServlet

Als übergeordnetes Servlet mit einigen grundlegenden Funktionalitäten dient die Klasse MCRServlet. Die Hauptaufgabe von MCRServlet ist dabei die Herstellung der Verbindung zur Sessionverwaltung (siehe Die Session-Verwaltung). Das Zusammenspiel der relevanten Klassen ist im folgenden Klassendiagramm (Abbildung) verdeutlicht.

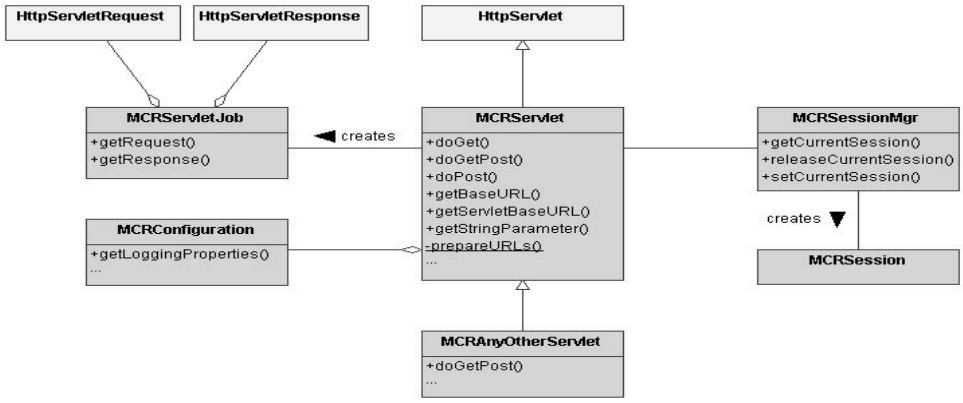


Abbildung 3.4: Klassendiagramm Common Servlets

Wie an anderen Stellen im MyCoRe-System auch, kann auf Konfigurationsparameter wie zum Beispiel den Einstellungen für das Logging über das statische Attribut MCRConfiguration zu gegriffen werden. Dies wird ausführlich ineinem anderen Kapitel beschrieben.

MCRServlet selbst ist direkt von HttpServlet abgeleitet. Sollen andere Servlets im MoCoRe-Softwaresystem die von MCRServlet angebotenen Funktionen automatisch nutzen, so mussen sie von MCRServlet abgeleitet werden. Im Klassendiagramm ist das durch die stellvertretende Klasse MCRAnyOtherServlet angedeutet. Es wird empfohlen, dass die abgeleiteten Servlets die Methoden doGet() und doPost() nicht überschreiben, denn dadurch werden bei einem eingehenden Request auf jeden Fall die Methoden von MCRServlet ausgeführt.

Der Programmablauf innerhalb von MCRServlet ist im folgenden Sequenzdiagramm (sihe Abbildung) dargestellt. Bei einem eingehenden Request (doGet() oder doPost()) wird zunächst an MVRServlet.doGetPost() delegiert.[8]

8 Bei dieser Delegation wird ein Parameter mit geführt, über den feststellbar ist, ob es sich um einen GET- oder POST-Request gehandelt hat.

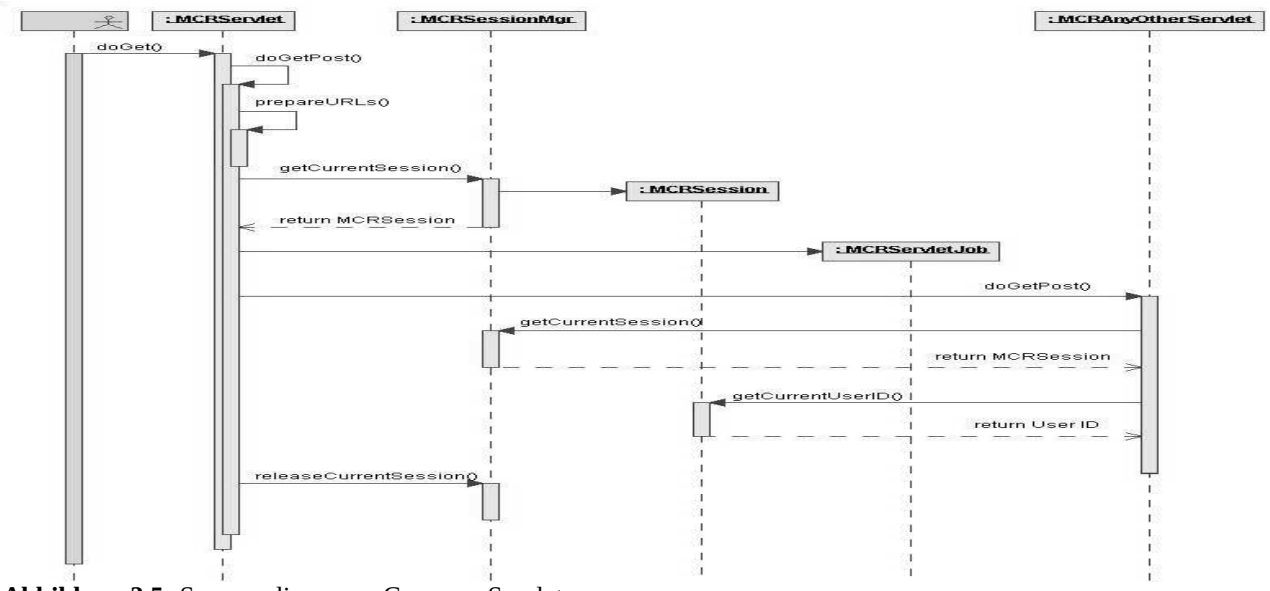


Abbildung 3.5: Sequenzdiagramm Common Servlets

Falls nicht schon aus vorhergehenden Anfragen an das MCRServlet bekannt, werden in doGetPost () die Base-URL und die Servlet-URL des Systems bestimmt. Dabei besteht die Servlet-URL aus der Base-URL und dem angehängten String 'servlets/'. Darauf folgend wird die für diese Session zugehörige Instanz von MCRSession bestimmt. Das Verfahren dazu ist im Ablaufdiagramm dargestellt.

Die Session kann bereits durch vorhergehende Anfragen existieren. Falls dies der Fall ist, kann das zu gehörige Session-Objekt entweder über eine im HttpServletRequest mitgeführte SessionID identifiziert oder direkt der HttpSession entnommen werden. Existiert noch keine Session, sowird ein neues Session-Objekt über den Aufruf von MCRSessionMgr.getCurrentSession() erzeugt. Nachfolgend wird das SessionObjekt an den aktuellen Thread gebunden und zusätzlich in der HttpSession abgelegt.

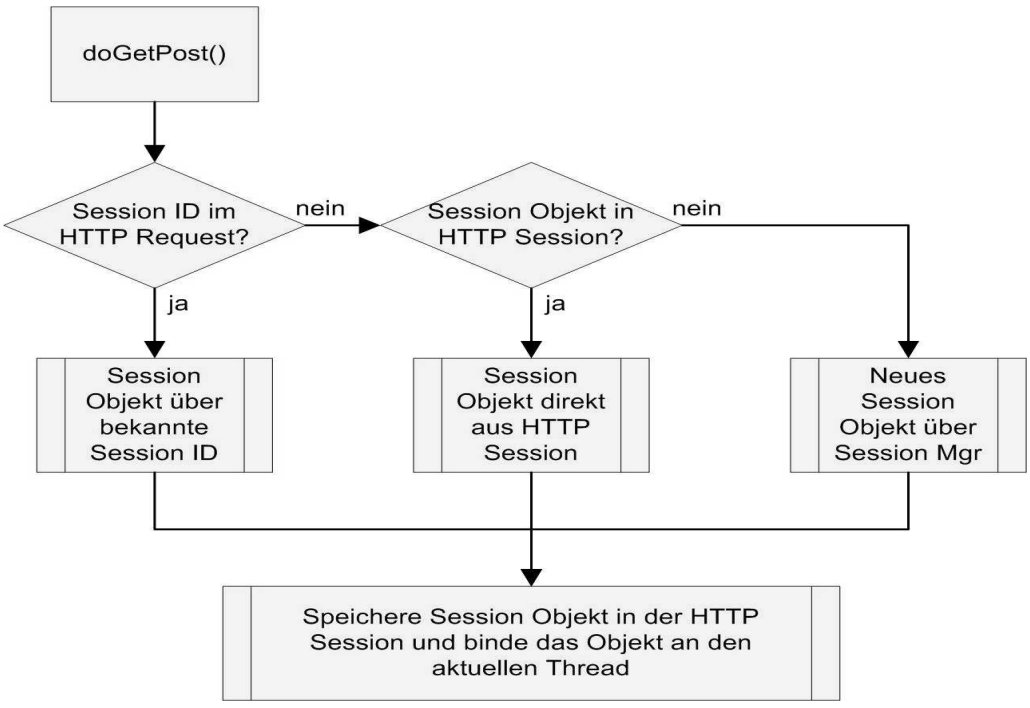


Abbildung 3.6: Ablaufdiagramm für MCRServlet.doGetPost()

Im Sequenzdiagramm gehen wird avonaus, dass die Sitzung neu ist und deswegen ein Session-Objekt über MCRSessionMgr.getCurrentSession() erzeugt werden muss. Schliesslich wird eine Instanz von MCRServletJob erzeugt. Diese Klasse ist nichts weiter als ein Container für die aktuellen HttpServletRequest und HttpServletResponse Objekte und hat keine weitere Funktionalität (siehe Klassendiagramm).[9]

An dieser Stelle wird der Programmfluss an das abgeleitete Servlet (in diesem Beispiel MCRAnyOtherServlet) delegiert. Dazu muss das Servlet eine Methode mit der Signatur

```
public void doGetPost(MCRServletJob job) {}
```

implementieren. Wie das Sequenzdiagramm beispielhaft zeigt, kann MCRAnyOtherServlet danach gegebenenfalls auf das Session-Objekt und damit auf die Kontextinformationen zugreifen. Der Aufruf an den SessionManager dazu wäre:

```
MCRSession mcrSession=MCRSessionMgr.getCurrentSession();
```

Es sei bemerkt, dass dies nicht notwendigerweise genau so durch geführt werden muss. Da wegen den geschilderten Problemen mit threadlocal Variablen in Servlet-Umgebungen das Session-Objekt auch in der HttpSession abgelegt sein muss, könnte man die Kontext informationen auch aus der übergebenen Instanz von MCRServletJob gewinnen.

3.4.3 Das Login-Servlet und MCRSession

Das LoginServlet, implementiert durch die Klasse MCRLoginServlet, dient zum Anmelden von Benutzern und Benutzerinnen über ein We-Formular. Die Funktionsweise ist wie folgt: Wie in Abschnitt 3.7.2 empfohlen, überschreibt MCRLoginServlet nicht die von MCRServlet geerbten Standard-Methoden doGet() und doPost(). Meldet sich ein Benutzer oder eine Benutzerin über das MCRLoginServlet an, so wird dahe rzunächst die Funktionalität von MCRServlet ausgenutzt und die in Abschnitt3.7.2 beschriebene Verbindung zur Sessionverwaltung herstellt. Wie dort ebenfalls beschrieben, wird der Programmfluss an das Login-Servlet über die Methode MCRLoginServlet.doGetPost() delegiert. Der Ablauf in doGetPost() wird imfolgenden Diagramm dargestellt und ist selbsterklärend.

Der resultierende XML Outpu-Stream muss vom zugehörigen Stylesheet verarbeitet werden und hat die folgende Syntax:

9 Das Speichern des Session-Objekts in der HttpSession ist notwendig, weil in einer typischen Servlet-Engine mit Thread-Pool Umgebung nicht davon ausgegangen werden darf, dass bei aufeinanderfolgenden Anfragen aus dem selben Kontext auch der selbe Thread zugeweisen wird.

```
<mcruser unknownuser="true| | false" userdisabled="true| | false"    invalidpassword="true| false">
<guestid>...</guestid>
<guestpwd>...</guestpwd>
<url>...<url>
</mcruser>
```

Abbildung 3.7: XML Output des LoginServlets

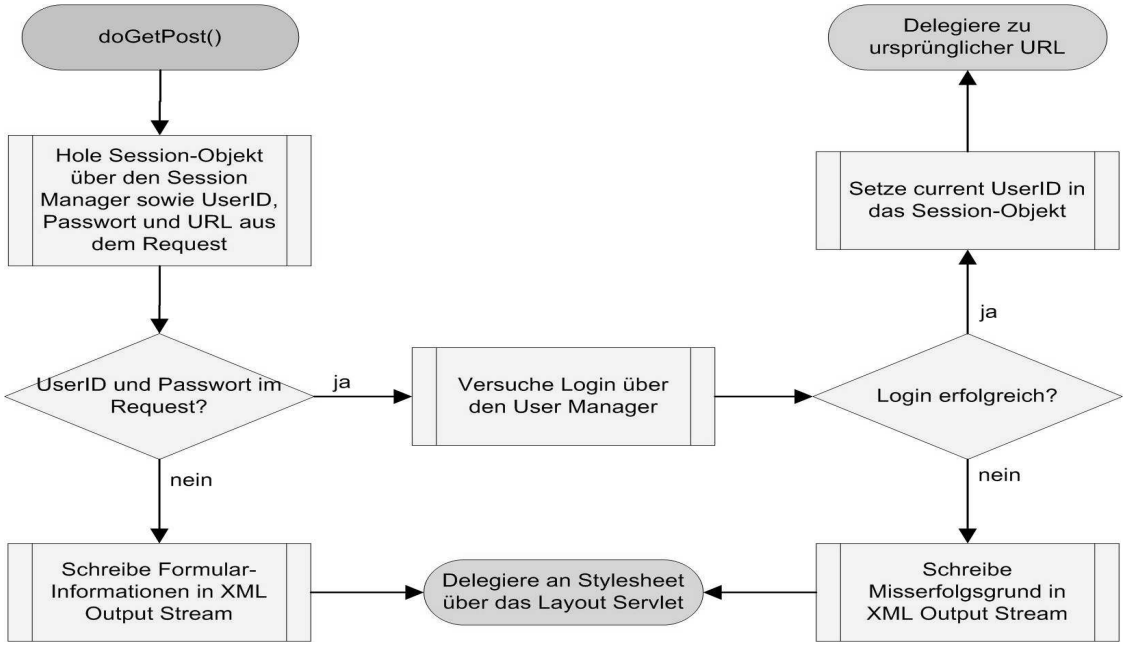


Abbildung 3.8: Ablaufdiagramm für MCRLoginServlet.doGetPost()

Bei einer missglückten Anmeldung wird der Grund dafür in Form eines Attributes auf true oder false gesetzt. Das Stylesheet kann dann die entsprechende Meldung ausgeben. Die GastUser-ID und das GastPasswort werden aus einer Konfigurationsdatei gelesen. Die URL schließlich wird dem HttpRequest entnommen und sollte dort von der aufrufenden Seite bzw. vom aufrufenden Servlet gesetzt sein. Ist sie nicht gesetzt, so wird die Base-URL des MyCoRe-Systems verwendet.

3.4.4 Generieren von Zip-Dateien

Das Zip-Servlet, implementiert durch die Klasse MCRZipServlet, dient dem Ausliefern der Derivate und der Objektmetadaten als gepackte Zip-Datei. Aus der Konfigurationsdatei mycore.properties.zipper holt sich das Servlet über die Variable MCR.zip.metadata.transformer den Namen des Stylesheets, welches die Metadatentransformation in das gewünschte Auslieferungsformat vornimmt. Im Mycoresample verwenden wir hierfür Qualified Dublin Core.

Aufrufmöglichkeiten des Servlets:

$ServletsBaseURL/MCRZipServlet?id=MCRID

$ServletsBaseURL/MCRZipServlet?id=MCRID/foldername

MCRID ist die ID eines Objekts vom Typ <mycoreobject> oder <mycorederivate>. Im Fall von <mycoreobject> werden die Dateien aller dem Objekt zugeordneten Derivate und ein XML-File mit den Metadaten des Objekts zusammengepackt. Im Fall von <mycorederivate> werden alle Dateien des angegebenen Derivats zusammengepackt. Die Option MCRID/foldername ist nur zulässig, wenn MCRID ein Objekt vom Typ <mycorederivate> bezeichnet. Dann wird nur der mit „foldername“ angegebene Ordner des betreffenden Derivats gezippt.

Wer geschützte Inhalte anbietet, sollte das Zip-Servlet erst dann in seine Anwendung integrieren, wenn die Zugriffskontrolle in Mycore gewährleistet werden kann. Dies ist momentan (04.2005) noch nicht der Fall, das Zip-Servlet läßt sich mit jeder MCRID aufrufen.

3.5 XML Funktionalität

3.5.1 URI Resolver

Die Klasse org.mycore.common.xml.MCRURIResolver implementiert einen Resolver, mit dem an verschiedenen Stellen im MyCoRe System XML-Daten über URIs gelesen werden können. Der Resolver wird zur Zeit an folgenden Stellen eingesetzt:

- Bei der Verarbeitung von Stylesheets im LayoutServlet, wenn XML-Daten über die XSL-Funktion document( ) in ein Stylesheet nachgeladen werden oder wenn ein untergeordnetes Stylesheet mittels xsl:include nachgeladen wird.
- Beim Import von Editor-Definitionsteilen mittels des include-Elementes des Editor Frameworks.

Der Resolver unterstützt die folgenden Schemata bzw. Protokolle:

file://[Pfad]
liest eine statische XML-Datei vom Dateisystem des Servers
Beispiel: file:///usr/local/tomcat/conf/server.xml liest die Datei /usr/local/tomcat/conf/server.xml

webapp:[Pfad]
liest eine statische XML-Datei vom Dateisystem der Webapplikation. Im Gegensatz zur file Methode kann der Pfad der zu lesenden Datei relativ zum Wurzelverzeichnis der Webapplikation angegeben werden. Der Zugriff erfolgt direkt, d. h. ohne HTTP Request oder Anwendung eines Stylesheets.
Beispiel: webapp:config/labels.xml

http://[URL]
https://[URL]
liest eine XML-Datei von einem lokalen oder entfernten Webserver

request:[Pfad]
liest eine XML-Datei durch einen HTTP Request an ein Servlet oder Stylesheet innerhalb der aktuellen MyCoRe-Anwendung. Im Gegensatz zur http/https Methode ist der Pfad relativ zur Basis-URL der Webapplikation anzugeben, die MCRSessionID wird automatisch als HTTP GET Parameter ergänzt.
Beispiel: request:servlets/MCRQueryServlet?XSL.Style=classif-to-items&type=class&hosts=local &lang=de&query=/mycoreclass[@ID='DocPortal_class_00000002']

resource:[Pfad]
liest eine XML-Datei aus dem CLASSPATH der Webapplikation, d. h. die Datei wird zunächst im Verzeichnis WEB-INF/classes/ und als nächstes in einer der jar-Dateien im Verzeichnis WEB-INF/lib/ der Webapplikation gesucht. Diese Methode bietet sich an, um statische XML-Dateien zu lesen, die in einer jar-Datei abgelegt sind.
Beispiel: resource:ContentStoreSelectionRules.xml

session:[Key]
liest ein XML Element, das als JDOM-Element in der aktuellen MCRSession abgelegt ist. Mittels der put() Methode der Klasse MCRSession kann analog zu einer Java Hashtable unter einem Schlüssen ein Objekt abgelegt werden. Ein Servlet kann so z. B. ein JDOM Element in der MCRSession ablegen, den Schlüssel einem Stylesheet über einen XSL Parameter mitteilen. Der MyCoRe Editor kann dieses JDOM Element dann mittels der get() Methode aus der Session lesen.
Beispiel: session:mylist liest das JDOM XML Element, das als Ergebnis von MCRSessionMgr.getCurrentSession().get("mylist"); zurückgegeben wird.

Bei der Verarbeitung von include-Anweisungen in Editor-Definitionen dürfen die folgenden URI Schemata verwendet werden:

file webapp http https request resource session

Beim Aufruf der xsl document() Funktion innerhalb eines MCR Stylesheets können die folgenden URI Schemate verwendet werden:

file webapp http https resource session

3.6 Das MyCoRe Editor Framework

3.6.1 Funktionalität

Das Metadatenmodell einer MyCoRe Anwendung ist frei konfigurierbar. Dementsprechend benötigt ein MyCoRe System auch einen Online-Editor für diese Metadaten, der frei konfigurierbar ist. Aus dieser Anforderung heraus entstand das MyCoRe Editor Framework, das aus einem XSL Stylesheet und einer Menge von Java-Klassen besteht.

Verschiedene MyCoRe Anwendungen können über XML-Definitionsdateien nahezu beliebige Online-Eingabemasken für Metadaten gestalten. Das Framework verarbeitet diese Editor-Definitionsdateien und generiert daraus den HTML-Code der Webseite, die das Online-Formular enthält. Nach Abschicken des Formulars generiert das Framework aus den Eingaben ein dem Metadatenmodell entsprechendes XML-Dokument, das an ein beliebiges Servlet zur endgültigen Verarbeitung (z. B. zur Speicherung) weitergereicht wird. Ebenso können existierende XML-Dokumente als Eingabe in die Formularfelder des Editors dienen, so dass sich vorhandene Metadaten bearbeiten lassen. Das Framework regelt dabei die Abbildung zwischen den XML-Elementen und –Attributen und den Eingabefeldern der resultierenden HTML-Formularseite, indem es die in der Editor-Definitionsdatei hinterlegten Abbildungsregeln verarbeitet.
Inzwischen ist das Editor Framework auch in der Lage, einzelne Dateien zusammen mit den Formulareingaben in das System hochzuladen und zur Weiterverarbeitung an ein Servlet durchzureichen. Die Validierung der Eingabefelder ist strukturell vorbereitet, aber derzeit noch

nicht implementiert. Prinzipiell erlaubt das Editor Framework, beliebige XML-Dokumente in HTML-Formularen zu erzeugen oder zu bearbeiten.

3.6.2 Architektur

Die folgende Abbildung zeigt die Architektur des MyCoRe Editor Frameworks:

Bild folgt

Ein HTML Formular, das man mit Hilfe des Frameworks realisieren möchte, wird im folgenden Editor genannt. Jeder Editor besitzt eine eindeutige ID (z. B. **document**)[10] und eine Definitionsdatei im XML-Format (*editor-document.xml*). In dieser Definitionsdatei ist festgelegt, aus welchen Eingabefeldern in welcher optischen Anordnung der Editor besteht und wie die Abbildung zwischen den Eingabefeldern und der zugrundeliegenden XML-Darstellung der Daten aussieht.

Ein Editor ist üblicherweise in eine umgebende Webseite eingebunden, die das Formular enthält. Die umgebenden Webseiten referenzieren über die Editor ID den Editor, der an einer bestimmten Stelle der sichtbaren Webseite eingebunden werden soll. Der HTML-Code der Webseite selbst wird in einem MyCoRe System ja aus einem beliebigen XML-Dokument (z. B. *anypage.xml*)[11] mittels eines dazu passenden XSL Stylesheets (*anypage.xsl*)[12] generiert. Um in eine solche Webseite einen Editor integrieren zu können, muss *anypage.xml* ein XML-Element enthalten, das auf den einzubindenden Editor verweist , zusätzlich muss *anypage.xsl* das Stylesheet *editor.xsl* einbinden. Das Stylesheet verarbeitet die Referenz auf den Editor und integriert den HTML-Code des Formulars in die umgebende Webseite. So ist es möglich, Editor-Formulare in beliebige Webseiten einzubinden, unabhängig von ihrem Layout oder ihrer Struktur.

```
<editor id="document">
<include uri="webapp:editor/editor-document.xml"/>
</editor>
```

Abbildung 3.9: : Einbindung des Editors in eine Webseite

Das Stylesheet editor.xsl aus dem Framework verarbeitet die Definition in editor-document.xml und erzeugt daraus eine HTML-Tabelle mit den entsprechenden Beschriftungen und Formularfeldern für die Eingabe der Metadaten. Es lassen sich jedoch nicht nur neue Daten eingeben, auch existierende Metadatenobjekte können bearbeitet werden. Das Stylesheet kann dazu von einer beliebigen URL ein XML-Dokument einlesen, verarbeitet dieses dann entsprechend den Abbildungsregeln aus editor-person.xsl und füllt die Eingabefelder des Formulars mit den Inhalten der XML-Elemente und –Attribute.

Wenn der Benutzer im Browser die Eingaben in die Formularfelder getätigt hat und das Formular abschickt, werden die Eingaben durch das Servlet MCREditorServlet aus dem Framework verarbeitet. Das Servlet generiert aus den Eingaben ein XML-Dokument, entsprechend den Regeln aus editor-document.xml. Dieses XML-Dokument wird anschließend an ein beliebiges anderes Servlet weitergereicht, welches in der Formulardefinition angegeben ist und welches dann die Daten

10 In den MyCoRe-Applikationen meißt eine Mischung aus dem MCRObjectID-Typ und der Bezeichnung eines Verarbeitungsschrittes (editor-commit-document.xml).
11 z. B. editor_form_...xml im MyCoReSample
12 z. B. MyCoReWePage.xsl im MyCoreSample

endgültig verarbeiten kann und z. B. das XML-Dokument als MyCoRe Metadaten-Objekt speichert. MCREditorServlet übergibt dabei dem Ziel-Servlet (z. B. **AnyTargetServlet**) ein Objekt vom Typ MCREditorSubmission, das nicht nur das XML-Dokument, sondern auch eventuell gemeinsam mit den Formulardaten übertragene Dateien als InputStream enthält. Damit ist es möglich, auch einfache Datei-Upload-Formulare über das Editor Framework zu realisieren.

Das Ziel-Servlet erhält zusätzlich auch die ursprünglichen HTTP Request Parameter aus dem abgeschickten Formular, so dass es bei Bedarf auch direkt auf die Werte bestimmter Eingabefelder zugreifen kann und nicht zwingend unbedingt das resultierende XML-Dokument beachten muss. Dadurch ist es möglich, das Editor Framework auch mit bereits existierenden Servlets zu nutzen, die bisher keine XML-Dokumente als Eingabe verarbeiten. Das Framework ist daher auch in der Lage, die Eingaben an eine beliebige URL statt an ein Servlet zu senden, so dass das Ziel der Eingaben an sich auch ein externes CGI-Skript oder ein Servlet in einem entfernten System sein könnte. Das schafft ein hohes Mass an Flexibilität, durch das das Editor Framework auch genutzt werden kann, um z. B. Suchmasken oder Login-Formulare zu erzeugen. Die Nutzung ist nicht allein auf die Realisierung von Metadaten-Editoren beschränkt.

3.6.3 Workarounds

Einige Funktionen sind derzeit noch nicht implementiert, es existieren aber Workarounds dafür:

- Eingabevalidierung:
  Soll: Die Editor-Definitionsdateien werden dafür Regeln enthalten, die nach Abschicken des Formulars durch MCREditorServlet überprüft werden.
  Workaround: Die Validierung kann im Ziel-Servlet erfolgen, entweder auf Basis der ursprünglichen Formularfelder oder auf Basis des daraus erzeugten XML-Dokumentes.
- Namespaces:
  Soll: Namespaces können in der Editor Definition deklariert werden, zusammen mit der Abbildung zwischen den Eingabefeldern und der XML-Darstellung der Daten.
  Workaround: Namespaces werden im Ziel-Servlet hinzugefügt, indem das generierte JDOM-Dokument entsprechend nachbearbeitet wird. Dies könnte auch ein Stylesheet tun.

3.6.4 Beschreibung der Editor-Formular-Definition

3.6.4.1 Die Grundstruktur

Im folgenden soll die Gestaltung eines Editor-Formulars näher beschrieben werden. Als Beispiel ist ein Formular für ein Dokument gedacht. Die Daten werden z. B. in der Datei editor-document.xml gespeichert.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id="document">
...
</editor>
```

Abbildung 3.10: Rahmen der Formular-Definition

Als nächstes definieren wir die Eingabefelder des Editors. Ein Editor besteht aus ein verschiedenen Komponenten, die innerhalb des Elements **components** deklariert werden. Dies können einfache Komponenten z. B. für Auswahllisten und Texteingabefelder sein, oder aber zusammengesetzte Panels, mit deren Hilfe man einfache Komponenten in einer Tabellengitterstruktur anordnen kann. Panels können wiederum andere Panels beinhalten, so dass sich komplexere Layouts erzeugen lassen. Jede Komponente besitzt eine innerhalb des Editors eindeutige ID, über die sie referenziert werden kann. Jeder Editor besitzt ein **Root-Panel**, die äußerste Struktur, mit der die Anordnung der Komponenten begonnen wird. Die ID dieses **Root-Panels** wird als Attribut im Element **components** angegeben.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id=“document“>
  <components root="document">
    <panel id="document" lines="on">
    </panel>
  </components>
</editor>
```

Abbildung 3.11: Definition des Root-Panels

Innerhalb eines Panels lassen sich andere Komponenten oder Panels anordnen. Solche Komponenten werden im einfachsten Fall einfach in die Gitterzellen eines Panels eingebettet. Jede Zelle entspricht einem **cell** Element innerhalb des Panels. Jede Zelle besitzt eine **row**- und **col**-Koordinate, mit deren Hilfe die Zelleninhalte von links nach rechts in Spalten (**col**) und von oben nach unten in Zeilen (row) angeordnet werden. Dabei ist für Textfelder noch die Angabe der Sprache möglich.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id=“document“>
  <components root="document">
    <panel id="document" lines="on">
      <cell row="1" col="1">
        <text><label xml:lang=“de“>Titel:</label></text
      </cell>
      <cell row="1" col="2">
        <textfield width="80" />
      </cell>
      <cell row="2" col="1">
        <text><label xml:lang=“de“>Autor:</label></text
      </cell>
      <cell row="2" col="2">
        <textfield width="80" />
      </cell>
    </panel>
  </components>
</editor>
```

Abbildung 3.12: Definition eines einfachen Formulares

Innerhalb der Gitterzellen füllen die Komponenten in der Regel nicht den gesamten Platz aus, daher können sie in ihrer Zellen mittels des anchor Attributs anhand der Himmelsrichtungen (NORTH, NORTHEAST, EAST, ... bis CENTER) angeordnet werden. Wir ordnen die Beschriftungen rechtsbündig, die Texteingabefelder linksbündig an. Außerdem wird nun in der untersten rechten Zelle rechtsbündig einen "Speichern" Button zum Absenden des Formulars ergänzt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id=“document“>
  <components root="document">
    <panel id="document" lines="on">
      <cell row="1" col="1" anchor="EAST">
        <text><label xml:lang=“de“>Titel:</label></text
      </cell>
      <cell row="1" col="2" anchor="WEST">
        <textfield width="80" />
      </cell>
      <cell row="2" col="1" anchor="EAST">
        <text><label xml:lang=“de“>Autor:</label></text
      </cell>
      <cell row="2" col="2" anchor="WEST">
        <textfield width="80" />
    </cell>
     <cell row="3" col="2" anchor="EAST">
       <submitButton width="100px" >
         <label xml:lang="de">[Speichern]</label>
      </submitButton>
     </cell>
    </panel>
  </components>
</editor>
```

Abbildung 3.13: Definition der Ausrichtung und des Submit-Button

Im nächsten Schritt soll nun die Definition aus dem Root-Panel ausgegliedert werden. Sie verwenden dazu ein eigenes Panel. Diese Technik wird verwendet, um Panels mehrfach zu nutzen und den Quellcode des Formulars zu strukturieren. dabei wird gezeigt, wie sich Panels untereinander aufrufen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id="document">
  <components root="document">
    <panel id="document" lines="on">
     <cell row="1" col="1" anchor="EAST" ref="form">
     <cell row="2" col="1" anchor="EAST">
       <submitButton width="100px" >
         <label xml:lang="de">[Speichern]</label>
       </submitButton>
     </cell>
   </panel>

    <panel id="form">
      <cell row="1" col="1" anchor="EAST">
        <text><label xml:lang="de">Titel:</label></text
      </cell>
      <cell row="1" col="2" anchor="WEST">
        <textfield width="80" />
      </cell>
      <cell row="2" col="1" anchor="EAST">
        <text><label xml:lang="de">Autor:</label></text
      </cell>
      <cell row="2" col="2" anchor="WEST">
        <textfield width="80" />
      </cell>
    </panel>
  </components>
</editor>
```

Abbildung 3.14: Auslagen von Definitionen aus dem Root-Panel

Soll das Panel **form** nun mehrfach genutzt werden, z. B. in mehreren Formularen, so ist es sinnvoll das Panel in eine gemeinsame Definitionsdatei *imports-common.xml* auszulagern. Im eigentlichen Formular wird dann nur noch auf das form-Panel verwiesen. Die Einbindung der ausgelagerten Panels erfolgt mit der **include**-Anweisung. Die einzufügende Datei wird nun im System mittels **EntityResolver** gesucht und eingebunden. Auf diese Weise lassen sich elegante und wartungsarme Formulare gestalten.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id="document">
 <components root="document">
  <include uri="webapp:editor/imports-common.xml" />
  <panel id="document" lines="on">
   <cell row="1" col="1" anchor="EAST" ref="form">
   <cell row="2" col="1" anchor="EAST">
     <submitButton width="100px" >
       <label xml:lang="de">[Speichern]</label>
     </submitButton>
    </cell>
  </panel>
 </components>
</editor>
```

Abbildung 3.15: Auslagen von Definitionen aus dem dem Editor-Formular

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE imports>

<imports>
   <panel id="form">
    <cell row="1" col="1" anchor="EAST">
      <text><label xml:lang="de">Titel:</label></text
    </cell>
    <cell row="1" col="2" anchor="WEST">
      <textfield width="80" />
    </cell>
    <cell row="2" col="1" anchor="EAST">
      <text><label xml:lang="de">Autor:</label></text
    </cell>
    <cell row="2" col="2" anchor="WEST">
      <textfield width="80" />
    </cell>
  </panel>
</imports>
```

Abbildung 3.16: Die imports-Formular-Definition

3.6.4.2 Abbildung zwischen Eingabefeldern und XML-Strukturen

Als nächstes definieren wir, wie die Eingabefelder auf XML-Elemente abgebildet werden. Dies geschieht über das Attribut var verschiedener Elemente der Editor-Definition. Zunächst wird im var Attribut des components Elements der Name des Root Elements der XML-Darstellung (**document**) festgelegt. Die Abbildung zwischen Eingabefeldern oder Panels und ihrer Zuordnung zu XML Elementen geschieht über das **var** Attribut in den **cell** Elementen, welche die Eingabekomponente enthalten.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id="document">
 <components root="document" var="/document">
  <include uri="webapp:editor/imports-common.xml" />
  <panel id="document" lines="on">
   <cell row="1" col="1" anchor="EAST" ref="form">
   <cell row="2" col="1" anchor="EAST">
     <submitButton width="100px" >
       <label xml:lang="de">[Speichern]</label>
     </submitButton>
   </cell>
  </panel>
 </components>
</editor>
```

Abbildung 3.17: Einfügen des XML-Main-Tag-Namen

Es gibt für Tags und Attribute innerhalb der Root-Komponente verschiedene Möglichkeiten der Gestaltung des Ausgabe-XML-Baumes. Die einfachste Art ist die direkte Zuordnung. Hier werden die Werte des Eingangs-XML-Files dem Feld in der Editormaske zugeordenet. weiterhin kann dem Feld noch ein Standardwert durch das Attribut **default** mitgegeben werden. Weiterhin können Felder, welche von der Dateneingabe ausgenommen werden sollen, deren Inhalte aber an die Ausgabeseite durchgereicht werden sollen, als **hidden**-Felder mitgeführt werden. Auch hier können Standardwerte angegeben werden, welche eingesetzt werden, wenn keine Daten für das Tag/Attribut vorhanden sind.

```
  <cell row="1" col="2" anchor="WEST">
   <textfield width="80" var="dc/title" default="Mein Buch" />
  </cell>
  <hidden var="dc/title/@lang" default="de" />
```

Abbildung 3.18: Integration von einfachen XML-Tags

Sollten Sie mehrere gleiche XML-Strukturen im Editor zur Bearbeitung anbieten wollen, so wird dies über eine Integer-Zahl in Klammer organisiert. Achtung, es werden immer nur soviel XML-Elemente übernommen und an die Ausgabe weitergereicht, wie angegeben sind. Um eine beliebige Anzahl zu präsentieren und zu bearbeiten nutzen sie die weiter unten angegeben Möglichkeit des Repeaters. Für hidden-Felder gibt es noch die Variante, mittels des **descendants="true"** - Attributs alle Kindelemente, Attribute und Wiederholungen des Elementes an den Ausgang durchzureichen.

```
<cell row="1" col="2" anchor="WEST">
  <textfield width="80" var="dc/title" default="Mein Buch" />
</cell>
<hidden var="dc/title/@lang" default="de" />
<cell row="2" col="2" anchor="WEST">
  <textfield width="80" var="dc/title[2]" default="" />
</cell>
<hidden var="dc/title[2]/@lang" default="de" />
<hidden var="dc/source" descendants="true" />
```

Abbildung 3.19: Integration von mehrfachen XML-Tags

3.6.4.3 Integration und Aufruf des Editor Framework

bereits zum Anfang des Kapitels wurde ein einfaches Syntax zum Aufruf des Editor Framework vorgestellt. Dieser Abschnitt soll nun umfassend zu diesem Thema informieren.

```
<editor id="document">
<include uri="webapp:editor/editor-document.xml"/>
</editor>
```

Abbildung 3.20: : Einbindung des Editors in eine Webseite

Das Editor Framework ist so konzipiert, dass es als XML-Sequenz in einer durch das MCRLayoutServlet darzustellenden Webseite integriert werden kann. D.h. Sie codieren eine XML Seite welche durch das Servlet und die zugehörigen XSLT-Stylesheets nach HTML konvertiert werden. Durch einen Include der Editor Framework XSLT-Dateien in das Layout zum Erzeugen der HTML-Seiten ermoglichen Sie die Einbindung ihrer Editor-Formulare. In der oben gezeigten XML-Sequenz wird jedoch nur die Definitionsdatei des Formulars beschrieben: „nimm aus der Web-Applikation die Datei *editor-document.xml* aus dem Verzeichnis *editor*. dabei ist noch nichts über die Datenquelle bekannt. Hierfür gibt es noch einige Parameter, welche durch das Framework ausgewertet werden. Diese können sowohl direkt im Browser in der Aufrufsequenz oder über Parametereinstellungen in einem Servlet mitgegeben werden.

```
http://.../document.xml?XSL.editor.source.new=true&type=...
```

oder in einem Servlet als Codestück

```
String base = getBaseURL() + myfile;
Properties params = new Properties();
params.put( "XSL.editor.source.new", "true" );
params.put( "XSL.editor.cancel.url", getBaseURL()+cancelpage);
params.put( "type",mytype );
params.put( "step",mystep );
job.getResponse().sendRedirect(job.getResponse().encodeRedirectURL(
 (buildRedirectURL( base, params )));
```

Abbildung 3.21: Codesequenz zum Aufruf des Editor Framework

Start-Parameter für das Lesen der zu bearbeitenden XML-Quelle:

Über XSL Parameter kann beim Aufruf des Editors gesteuert werden, ob bzw. aus welcher Quelle die zu bearbeitenden XML-Daten gelesen werden. Die zu bearbeitenden XML-Daten werden von einer URL gelesen. Falls die URL mit „http://“ oder „https://“ beginnt, wird sie als absolute URL direkt verwendet, andernfalls wird die URL als relativ zur WebApplicationBaseURL (context root), dem Startpunkt der WebApplication, interpretiert. Es gibt die folgenden vier Optionen:

1. **XSL.editor.source.new=true**
   Es wird das Formular mit leeren Datenfeldern gestartet.
2. Statische URL eines XML-Dokumentes (z. B. eines Metadaten-Templates), die in der Editor-Definition als Top-Level-Element angegeben ist (d.h. auf gleicher Ebene wie die target-Deklaration), z. B.
   <source url=“templates/document.xml“ />
3. URL, die durch ein Servlet oder Stylesheet zur Laufzeit gebildet wird, und die beim Aufruf des Editors über einen XSL Parameter übergeben wird:
   XSL.editor.source.url=templates/document.xml
4. URL, die aus einer Kombination einer statischen URL und eines ID=Tokens gebildet wird, das beim Aufruf des Editors über einen XSL Parameter übergeben wird. Im folgenden Beispiel wird zur Laufzeit das Token XXX durch den Parameterwert 4711 ersetzt:
   XSL.editor.source.id=4711
   <source url=“servlets/SomeServlet?id=XXX“ token=“XXX“ />

Start-Parameter für die Definition des Abbrechen-Buttons:

XSL.editor.cancel.url / XSL.editor.cancel.id
Diese Parameter werden im Abschnitt zur Verwendung eines Cancel-Buttons (Abbrechen-Knopf) erläutert, sie steuern die URL, zu der bei Klick auf einen „Abbrechen“ Button zurückgekehrt wird.

Start-Parameter für die Weitergabe an das Zielservlet:

In der Regel werden die Eingaben nach der Bearbeitung mit dem Editor an ein Zielservlet gesendet (target type=“servlet“). Man kann diesem Zielservlet auch HTTP Parameter mitschicken, die der Editor bei Abschicken des Formulars mitsendet. So können Informationen an das Zielservlet weitergegeben werden, die nicht Teil des zu bearbeitenden XML sind, z. B. die Information, ob es

sich um einen Update- oder Create-Vorgang handelt, die Objekt-ID etc. Der Editor reicht automatisch alle HTTP Request Parameter an das Zielservlet durch, deren Name nicht mit XSL beginnt.

Beispiel:

http://.../editor-document.xml?XSL.editor.source.id=4711&action=update&mode=bingoBongo

3.6.4.4 Ausgabeziele

Das Editor-Framework ist auch hinsichtlich der Verarbeitung der entstandenen Daten flexibel konfigurierbar. Wie bereits erwähnt, wird nach dem Betätigen des Sumit-Buttons das MCREditorServlet für eine erste Verarbeitung angestoßen. Diese reicht die Daten dann an ein weiteres konfigurierbares Ziel weiter. Welches das ist, wird mit der **target**-Zeile in der Formulardefinition festgelegt. Das Attribut **method** definiert, ob dazu HTTP GET oder POST verwendet wird, zu empfehlen ist in der Regel immer POST, was auch der Default-Wert ist und zwingend bei File Uploads gesetzt wird. Möglich sind:

- die Ausgabe als HTML-Seite zum Debugging des Ergebnisses:
  <target type=“debug“ method=“post“ format=“xml“ />
- die Weiterleitung der Daten in ein nachgeschaltetes Servlet im XML-Format:
  <target type=“servlet“ method=“post“ name=“MCRCheckDataServlet“ format=“xml“ />
- die Weiterleitung der Daten an ein nachgeschaltetes Servlet in der Form 'name=value', d. h. ohne Generierung eines XML-Dokumentes:
  <target type=“servlet“ method=“post“ name=“MCRCheckDataServlet“ format=“name=value“ />
- die Weiterleitung an LayoutServlet zur Anzeige des generierten XML-Dokumentes als Test:
  <target type=“display“ method=“post“ format=“xml“ />
- die Weiterleitung an eine beliebige URL als HTTP GET oder POST Request:
  <target type=“url“ method=“post“ format=“name=value“ url=“cgi-bin/ProcessMe.cgi“ />
- die Weiterleitung an einen anderen Editor, für den dieser Editor als „Subdialog“ fungiert. Dies wird in einem separaten Abschnitt erläutert.
  <target type=“subselect“ method=“post“ format=“xml“ />

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id=“document“>
<!--
<target type=“debug“ />
-->
<target type="servlet" name="MCRCheckDataServlet" method="post" format="xml" />
...
</editor>
```

Abbildung 3.22: Rahmen der Formular-Definition

Mit der nachfolgenden Java-Code-Sequenz kann nun auf die vom MCREditorServlet durchgereichten XML- und -File-Daten zugegriffen werden.

```
/**
 * This method overrides doGetPost of MCRServlet.<br />
 */
public void doGetPost(MCRServletJob job) throws Exception
  {
  // read the XML data
  MCREditorSubmission sub = (MCREditorSubmission)
    (job.getRequest().getAttribute("MCREditorSubmission"));
  org.jdom.Document indoc = sub.getXML();
  List files = sub.getFiles();

...
  }
```

Abbildung 3.23: Java-Code-Sequenz für den Zugriff

Neben der Übernahme der Ausgabewerte des MCREditorServlets als XML-Baum können diese auch in der Form 'name=value' durchgereicht werden. Hierzu müssen u. a. die **var**-Attribute entsprechend in eine einfache Form gebracht werden. Der Ausschnitt der Editor-Definition soll das verdeutlichen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE editor>

<editor id="document">
 <target type="servlet" name="MCRStartEditorServlet" method="get"
format="name=value" />
 <components root="formular">
  <panel id="document" lines="on">
   <cell row="1" col="1" anchor="EAST">
    <text><label xml:lang="de">Label:</label></text
   </cell>
   <cell row="1" col="2" anchor="WEST" var="label">
    <textfield width="80" />
   </cell>
...
 </components>
</editor>
```

Abbildung 3.24: Rahmen der Formular-Definition mit name=value

3.6.5 Syntax der Formularelemente

3.6.5.1 Aufbaus einer Zelle

```
<cell var="..." col="..." row="..." anchor="..." ref="..." sortnr="..." />
oder
<cell var="..." col="..." row="..." anchor="..." sortnr="...">
 Formularfelddefinition
</cell>
```

Abbildung 3.25: Syntax des cell-Elements

Das **cell**-Element stellt ein elementares Gebilde innerhalb eines Formulargitters dar. Seine Position wird durch die Attribute **col**, **row** und **anchor** beschieben. **col** gibt dabei die relative Spaltenzahl, **row** die relative Zeilenzahl an. Mit **anchor** kann die Ausrichtung gesteuert werden. Mögliche Werte sind hier NORTHWEST, NORTH, NORTHEAST, WEST, CENTER, EAST, SOUTHWEST, SOUTH und SOUTHEAST. Mit dem Attribut **ref** kann auf ein mehrfach verwendbares Element via ID referenziert werden. Über das Attribut **sortnr** kann die Reihenfolge der zu generierenden XML-Daten für die Ausgabe geregelt werden. Dieses Attribut bezieht sich dabei auf das gesamte Formular.

3.6.5.2 Texteingabefelder

Ein einzeiliges Texteingabefeld wird mittels eines **textfield** Elementes erzeugt, ein mehrzeiliges mittels eines **textarea** Elementes. Das Attribut **width** gibt die Breite des Texteingabefeldes in Anzahl Zeichen an. Das Attribute **height** gibt für mehrzeilige Texteingabefelder die Anzahl Zeilen an. Das Attribut **maxlenght** gibt bei textfield Elementen die maximale Länge der Eingabe an.

Texteingabefelder können einen Vorgabewert enthalten, der wahlweise über ein Attribut oder ein Kindelement mit dem Namen **default** angegeben wird. Die Verwendung eines default Elementes statt eines default Attributes ist sinnvoll, wenn mehrzeilige default-Werte angegeben werden sollen.

```
<textfield id="tf.name" width="80" default="Standardtext" maxlength="200" / >
<textarea id="tf.anschrift" width="80" height="5">
 <default>Musterstrasse 69
4711 Musterstadt </default>
</textarea>
```

Abbildung 3.26: Syntax von textfield und textarea

Alternativ kann auch ein AutoFill-Wert über das Attribut bzw. Element **autofill** angegeben werden. Während **default**-Werte immer Teil der Eingabe werden, werden AutoFill-Werte nur dann als Eingabe betrachtet, wenn der Nutzer sie ergänzt oder ändert. Unveränderte AutoFill-Werte werden also ignoriert. Für alle Mitarbeiter der Universität sind z. B. Teile der Email-Adresse oder der Anschrift in den meisten Fällen identisch. Falls der Nutzer diese Angaben im Eingabefeld ändert oder ergänzt, wird dies als Eingabe betrachtet. Ansonsten wird der AutoFill-Wert ignoriert. Default-Werte dagegen werden immer als Eingabe betrachtet. Wichtig ist, dass textfield und textarea Elemente ein innerhalb des Editors eindeutiges ID Attribut besitzen.

```
<textfield id="tf.email" width="80" autofill="@uni-duisburg-essen.de" />
<textarea id="ta.anschrift" width="80" height="5">
 <autofill>Universitätsstr. 9-11
45141 Essen</autofill>
</textarea>
```

Abbildung 3.27: AutoFill-Werte in textfield und textarea

3.6.5.3 Passwort-Eingabefelder

Passwort-Eingabefelder werden über das Element password erzeugt. Sie können keine default- oder autofill-Werte besitzen. Bei der Eingabe werden Sternchen statt der eingegebenen Zeichen angezeigt. Das Attribut **width** gibt die Breite des Eingabefeldes in Anzahl Zeichen an. Leider ist es bei allen Browsern so, dass dieses Feld keine vorhandenen Inhalte bearbeiten kann, d. h. Passworte sind aus Sicherheitsgründen immer neu einzugeben.

```
<password width="10" />
```

Abbildung 3.28: Synatax des password-Elements

3.6.5.4 Einfache Checkboxen

Das Element checkbox generiert eine alleinstehende Checkbox, bei der über einen „Haken" ein Wert ein- oder ausgeschaltet werden kann. Das Attribut value gibt den Wert an, der gesetzt werden soll, wenn die Checkbox markiert ist. Das Attribut checked gibt mit den möglichen Werten true und false an, ob als default die Checkbox markiert sein soll oder nicht. Die Beschriftung der Checkbox erfolgt über ein label Attribut bzw. untergeordnete mehrsprachige label Element, vgl. dazu den Abschnitt zu Beschriftungen.

```
<cell ... var="@raucher">
  <checkbox value="true" checked="false" label="Ja, ich bin Raucher" />
</cell>
```

3.6.5.5 Auswahllisten

Auswahllisten werden über das Element **list** erzeugt. Die in der Liste darzustellenden Werte werden über geschachtelte **item** Elemente angegeben. Das Attribut **type** gibt an, in welchem Format die Listenwerte dargestellt werden. Der Typ **dropdown** stellt die Listenwerte als einzeilige Dropdown-Auswahlliste dar. Das Attribut **width** gibt dabei die Breite des Auswahlfeldes in CSS-Syntax an, d.h. es sind Angaben in Anzahl Zeichen oder Pixeln etc. möglich. Der Typ **multirow** stellt die Listenwerte als mehrzeiliges Auswahlfeld dar. Das Attribut **rows** gibt dabei die Anzahl Zeilen an, die maximal zu sehen sind. Enthält die Liste mehr Werte als Zeilen, werden Scrollbalken dargestellt. Das Attribut **multiple** mit den möglichen Werten true und false gibt an, ob eine Mehrfachauswahl möglich ist. Das Attribut **default** gibt ggf. eine Vorauswahl vor.

```
<list type="dropdown" width="100px" default="2" />
  <item value="1"><label xml:lang="de>eins</label></item>
  <item value="2"><label xml:lang="de>zwei</label></item>
  <item value="3"><label xml:lang="de>drei</label></item>
</list>
<list type="multirow" width="100px" rows="3" multiple="true">
  <item value="a"><label xml:lang="de>Fred</label></item>
  <item value="b"><label xml:lang="de>Willi</label></item>
  <item value="c"><label xml:lang="de>Otto</label></item>
</list>
```

Abbildung 3.29: Syntax einer Auswahlliste

Die in einer Liste dargestellten Werte werden über **item** Elemente angegeben. Diese können auch mehrstufig geschachtelt sein, um eine hierarchische Struktur darzustellen (d.h. item-Element können wiederum item-Element enthalten). Eine Schachtelung ist nur für die Listentypen **dropdown** und **multirow** erlaubt und sinnvoll. Die Hierarchie wird dabei automatisch durch Einrückungen dargestellt. Die Beschriftung der item-Element wird über geschachtelte **label**-Attribute oder Elemente definiert und kann mehrsprachig angegeben werden, vgl. dazu den folgenden Abschnitt zu Beschriftungen. Die Verwendung von <list type="multirow" multiple="true">, d. h. mehrzeiliger Auswahllisten mit Mehrfachauswahl ist nur sinnvoll, wenn das aktuelle „var" Attribut der Zelle auf ein Element verweist, denn Attribute sind ja grundsätzlich in XML nicht wiederholbar. Beispiel:

```
<cell ... var="programmiersprache">
  <list type="multirow" multiple="true" rows="3">
    <item label="Java" value="java" />
    <item label="C++" value="cpp" />
    <item label="Pascal" value="pas" />
  </list>
</cell>
```

erzeugt eine dreizeilige Mehrfachauswahllist, die ggf. in mehrfach generierte XML-Elemente „programmiersprache" resultiert.

3.6.5.6 Radio-Buttons und Checkbox-Listen

Der Typ **radio** stellt die Listenwerte als eine Menge von Radiobuttons dar, es kann also nur ein Wert ausgewählt werden. Der Typ **checkbox** stellt die Listenwerte als eine Menge von Checkboxes dar, in diesem Fall können mehrere Werte ausgewählt sein. Bei diesen beiden Typen werden alle Werte der Liste nacheinander in einer Tabellenstruktur mit einer gewissen Anzahl Zeilen und Spalten ausgegeben. Es kann entweder die Anzahl Zeilen (Attribut **rows**) oder die Anzahl Spalten (Attribut **cols**) dieser Tabelle angegeben werden, der jeweils nicht angegebene Wert folgt automatisch aus der Anzahl der Werte in der Liste. Der Spezialfall rows="1" entspricht also einer Anordnung aller Listenwerte von links nach rechts in einer Zeile, der Fall cols="1" entspricht einer Anordung von oben nach unten in nur einer Spalte.

```
<list type=“radio” rows=2“ default="new">
  <item value=“1“><label xml:lang=“de>eins</label></item>
  <item value=“2“><label xml:lang=“de>zwei</label></item>
  <item value=“3“><label xml:lang=“de>drei</label></item>
</list>
<list type=“checkbox“ cols=”3”>
  <item value=“a“><label xml:lang=“de>Fred</label></item>
  <item value=“b“><label xml:lang=“de>Willi</label></item>
  <item value=“c“><label xml:lang=“de>Otto</label></item>
</list>
```

Abbildung 3.30: Syntax von Radio-Button und Checkbox-Listen

Analog zu multirow/multiple Listen ist die Verwendung von Checkbox-Listen nur sinnvoll für die Abbildung auf XML-Elemente, da XML-Attribute nicht wiederholbar sind (vgl. vorangehenden Abschnitt).

3.6.5.7 Beschriftungen

Beschriftungen von Eingabefeldern lassen sich mit den Elementen **text** und **label** definieren. Das Element **text** erzeugt eine frei platzierbare Beschriftung etwa für eine Texteingabefeld. Beschriftungen dürfen auch XHTML Fragmente enthalten, etwa um eine fette, kursive oder mehrzeilige Darstellung zu erreichen. Mehrsprachige Beschriftungen von Elementen können über mehrere **label** Elemente realisierte werden, wobei das Attribut **xml:lang** die Sprache angibt.

Die aktive Sprache wird wie bei anderen MyCoRe Stylesheets über den **XSL.Lang** Parameter definiert. Wenn ein **label** existiert, bei dem **xml:lang** der aktiven Sprache entspricht, wird dieses ausgegeben. Ansonsten, wird das **label** der Default-Sprache (Parameter **XSL.DefaultLang**) ausgegeben, falls es existiert. Ansonsten wird das Label ausgegeben, das im label Attribut oder im ersten **label** Element enthalten ist, das kein **xml:lang** Attribut enthält. Ansonsten wird das erste label Element verwendet, das überhaupt existiert.

```
<text>
  <label xml:lang=“de“><b>Eins</b></label>
  <label xml:lang=“en“><b>One</b></label>
</text>
```

Abbildung 3.31: Syntax eines Textfeldes

3.6.5.8 Abstandshalter

Ein Abstandshalter erzeugt leeren Raum als Platzhalter zwischen Elementen. Die Attribute **width** und **height** geben Breite und Höhe des Abstandes in CSS-Syntax, etwa in Anzahl Pixeln, an.

```
<space width="0px" height="50px" />
```

Abbildung 3.32: Syntax eines Abstandhalters

3.6.5.9 Hilfefenster

Ein Hilfefenster kann Hilfetext zu einem Eingabefeld im HTML-Format enthalten. Es wird als **[?]** Button im Formular dargestellt. Bei Anklicken des Hilfebuttons erscheint der Text in einem Popup-Fenster. Die Attribute **width** und **height** steuern die Breite und Höhe dieses Fensters. Wichtig ist, dass jedes helpPopup Element eine eindeutige ID besitzt. Optional kann der Hilfetext auch von einer externen URL geladen werden. Falls die URL mit „http://“ oder „https://“ beginnt, wird sie als absolute URL direkt verwendet, andernfalls wird die URL als relativ zur WebApplicationBaseURL (context root), dem Startpunkt der WebApplication, interpretiert.

```
<helpPopup id="url.help" width="400px" height="200px">
  In diesem Feld geben Sie bitte die <b>WWW Adresse (URL)</b>
  der Webseite ein. Bitte achten Sie darauf, dass die Adresse mit
  <i>http://</i> beginnt, da sonst der Link unter Umständen nicht funktioniert.
</helpPopup>

<helpPopup id="url.help" width="400px" height="200px" url="hilfe/url.html" />
```

Abbildung 3.33: Syntax eines Hilfetextes

3.6.5.10 Buttons

Über das **button** Element wird ein Knopf erzeugt, der bei Anklicken zu einer externen URL wechselt. Das Attibut **width** legt die Breite des Knopfes fest, das Attribut label oder ggf. mehrsprachige enthaltene **label** Elemente legen die Beschriftung des Knopfes fest. Falls die URL mit „http://“ oder „https://“ beginnt, wird sie als absolute URL direkt verwendet, andernfalls wird die URL als relativ zur WebApplicationBaseURL (context root), dem Startpunkt der WebApplication, interpretiert.

```
<button width="100px" url="index.html">
 <label xml:lang="de">Start</label>
</button>
```

Abbildung 3.34: Syntax eines einfachen Buttons

3.6.5.11 SubmitButton

Über das **submitButton** Element wird ein Knopf erzeugt, der bei Anklicken die Eingaben an das in der Konfiguration angegebene Ziel sendet. Das Attibut **width** legt die Breite des Knopfes fest, das Attribut label oder ggf. mehrsprachige enthaltene **label** Elemente legen die Beschriftung des Knopfes fest.

```
<submitButton width="100px">
 <label xml:lang="de">Start</label>
</submitButton>
```

Abbildung 3.35: Syntax des Submit-Buttons

3.6.5.12 CancelButton

Über das **cancelButton** Element wird ein Knopf erzeugt, der bei Anklicken die Bearbeitung des Formulars abbricht.[13] Das Attibut **width** legt die Breite des Knopfes fest, das Attribut label oder ggf. mehrsprachige enthaltene **label** Elemente legen die Beschriftung des Knopfes fest.

```
<cancelButton width="100px">
 <label xml:lang="de">Start</label>
</cancelButton>
```

Abbildung 3.36: Syntax des Cancel-Buttons

Die Ziel-URL des Buttons kann auf drei verschiedene Weisen gebildet werden. Falls die URL mit „http://“ oder „https://“ beginnt, wird sie als absolute URL direkt verwendet, andernfalls wird die URL als relativ zur WebApplicationBaseURL (context root), dem Startpunkt der WebApplication, interpretiert.

Statische URL, die in der Editor-Definition als Top-Level-Element angegeben ist (d.h. auf gleicher Ebene wie die target-Deklaration), z. B.
<cancel url="pages/goodbye.html" />

5. URL, die durch ein Servlet oder Stylesheet zur Laufzeit gebildet wird, und die beim Aufruf des Editors über einen XSL Parameter übergeben wird:
   XSL.editor.cancel.url=pages/goodbye.html
6. URL, die aus einer Kombination einer statischen URL und eines ID=Tokens gebildet wird, das beim Aufruf des Editors über einen XSL Parameter übergeben wird. Im folgenden Beispiel wird zur Laufzeit das Token XXX durch den Parameterwert 4711 ersetzt:
   XSL.editor.cancel.id=4711
   <cancel url="servlets/SomeServlet?id=XXX" token="XXX" />

3.6.5.13 Ausgabe von Werten aus dem Quelldokument

Das Element output kann bei der Bearbeitung einer existierenden XML-Quelle verwendet werden, um den Inhalt von Attributen oder Elementen im Formular als nicht bearbeitbaren Text auszugeben. Dabei ist zu beachten, dass der ausgegebene Wert bei Abschicken des Formulars nicht weitergegeben wird. Hierfür muss ggf. ein hidden Element verwendet werden. Das Attribut default gibt den Wert an, der ggf. ausgegeben wird wenn das Dokument keinen Wert enthält.

Beispiel:

<text><label>Dokument-ID:</label></text>

<output var="@id" default="(neu)" />

gibt den Wert des Attributes id aus, sofern dieses im XML Quelldokument existiert, ansonsten wird „(neu)“ ausgegeben.

13 Siehe auch **Integration und Aufruf des Editor Framework**

3.6.5.14 Wiederholbare Element mit Repeatern erstellen

Häufig sind einzeilne Eingabefelder oder ganz Panels wiederholbar. Das Element repeater schachtelt auf einfache Weise ein Eingabeelement oder ein ganzes Panel und macht dieses wiederholbar. Das var-Attribut der umgebenden Zelle muss auf ein Element verweisen, da Attribute nicht wiederholbar sind. Das Attribut min gibt an, wie oft minimal das wiederholte Element im Formular dargestellt wird, das Attribut max gibt die Maximalzahl an Wiederholungen an. In jedem Fall wird das Element minimal so oft dargestellt, wie es in der Eingabe auftritt.

Beispiele:

```
<cell ... var="creators/creator">
  <repeater min="1" max="10">
    <textfield id="tf.creator" width="80" />
  </repeater>
</cell>
```

wiederholt ein einzelnes Eingabefeld oder eine beliebige andere Komponente.

```
<cell ... var="creators/creator">
  <repeater min="3" max="6">
    <panel ...>
    </panel>
  </repeater>
</cell>
```

wiederholt ein ganzes Panel, dass wie hier im Beispiel auch direkt eingebettet definiert werden kann.

```
<cell ... var="creators/creator">
  <repeater min="3" max="6" ref="pcreator" />
</cell>
```

wiederholt die Komponente mit der ID pcreator.

Wiederholbare Elemente werden mit +/- Buttons dargestellt, mit denen neue Eingabefelder hinzugefügt bzw. dargestellte gelöscht werden können. Bei mehr als einer aktuell dargestellten Wiederholung werden Pfeile angezeigt, mit deren Hilfe die Reihenfolge der Elemente vertauscht werden kann.

3.6.5.15 Der Framework-interne FileUpload

Das Element file erlaubt es, eine einzelne Datei zusammen mit den Formulareingaben hochzuladen oder eine auf dem Server vorhandene Datei zu löschen oder zu aktualisieren. Vorausetzung ist hierfür die vollständige Integration des Upload-Service in den jeweiligen Java-Klassen.[14] Das Attribut **maxlenght** gibt optional die maximale Grösse der hochzuladenen Datei an. Der File Upload über ein HTML-Formular ist nur für relativ kleine Dateien mit wenigen MB zu empfehlen, andernfalls ist der in MyCoRe implementierte externe FileUpload zu nutzen. Das Attribut **accept** gibt optional den akzeptierten MIME Typ an. Ob diese Angaben beachtet werden, ist jedoch vom

14 Im MyCoReSample ist derzeit nur der Upload integriert.

Browser abhängig und daher nicht verlässlich. Die Darstellung eines Datei-Upload Feldes hängt ebenfalls sehr stark vom Browser ab und kann nicht über CSS gestaltet werden. In der Regel bestellt ein solches Feld aus einem Texteingabefeld (dessen Breite in Anzahl Zeichen das **width** Attribut angibt) und einem Button „Durchsuchen“. Beschriftung und Layout dieser Elemente sind nicht steuerbar und fest vom Browser vorgegeben, CSS Angaben funktionieren leider nicht zuverlässig. Über den Durchsuchen-Button kann eine Datei von der lokalen Platte gewählt werden, alternativ kann der Dateipfad im Texteingabefeld eingegeben werden. Mit Abschicken des Formulars wird der Dateiinhalt und der Dateiname an den Server übertragen, was unter Umständen lange dauert.

```
<cell row="1" col="1" anchor="WEST" var="pathes/path">
 <file width="60" maxlength="5000000" />
</cell>
```

Abbildung 3.37: Syntax des FileUpload

Zu beachten ist noch, dass für den FileUpload einige Property-Werte des Systems von bedeutung sind.

Dateien bis zu dieser Größe werden im Hauptspeicher des Servers gehalten:

MCR.Editor.FileUpload.MemoryThreshold=1000000

Dateien bis zu dieser Größe werden für HTTP Uploads akzeptiert:

MCR.Editor.FileUpload.MaxSize=5000000

Temorärer Speicherort für hochgeladene Dateien:

MCR.Editor.FileUpload.TempStoragePath=/tmp/upload

Das Auslesen der Dateien zur weiteren Verarbeitung kann z. B. wie folgt durchgeführt werden.

```
import org.apache.commons.fileupload.*;
…
public void doGetPost(MCRServletJob job) throws Exception
{
  MCREditorSubmission sub = (MCREditorSubmission)
    (job.getRequest().getAttribute("MCREditorSubmission"));
  List files = sub.getFiles();
  for( int i=0; i<files.size(); i++)
  {
    FileItem item = (FileItem)( files.get(i) );
    String fname = item.getName().trim();
…
    File fout = new File(dirname,fname);
    FileOutputStream fouts = new FileOutputStream(fout);
    MCRUtils.copyStream( item.getInputStream(), fouts );
    fouts.close();
  }
…
  }
```

Abbildung 3.38: Java-Code zum Lesen des FileUpload

Weitere Hinweise zum Auslesen der hochgeladenen Dateien finden Sie in der JavaDoc-Dokumentation der Klassen org.mycore.frontend.editor2.MCREditorSubmission und MCREditorVariable sowie in der Online-Dokumentation des Apache File Upload Paketes (http://jakarta.apache.org/commons/fileupload/).

Die Methode MCREditorSubmission.listFiles() liefert eine java.util.List von hochgeladenen FileItem Objekten. Die Apache Klasse FileItem besitzt Methoden getName() und getInputStream(), die den Dateinamen und den hochgeladenen Dateiinhalt liefern. Die Methode getFieldName() liefert den Pfad im XML-Dokument, zu dem die hochgeladene Datei gehört.

Alternativ kann auch über diesen Pfad auf eine bestimmte hochgeladene Datei direkt zugegriffen werden: Die Eingaben aus dem Editor werden als JDOM-Dokument an das Zielservlet übergeben und stehen über MCREditorSubmission.getXML() zur Verfügung. Mit JDOM-Operationen kann man nun zu dem JDOM-Element oder -Attribut navigieren, für das ggf. eine Datei im Formular hochgeladen wurde. Die Methode MCREditorSubmission.getFile() erwartet als Argument dieses JDOM-Objekt und liefert das dazu gehörende FileItem.

FileItem-Objekte sollten unmittelbar verarbeitet werden, da die Dateiinhalte nur temporär gespeichert sind. Der hochgeladene Inhalt kann mittels FileItem.write() in ein persistentes java.io.File kopiert werden oder in einem MCRFile gespeichert werden.

Zu beachten ist, dass File Uploads nicht in Editoren verwendet werden sollten, die Repeater enthalten. Will man mehr als eine Datei hochladen, sollte das file-Element „manuell“ mehrfach im Editor definiert werden, es darf jedoch kein Repeater verwendet werden.

3.6.5.16 Integration externer Datenquellen

Das Editor Framework gestattet die Einbindung externer Datenquellen in das Formular. Dies ermöglicht eine flexible Gestaltung von Eingabewerten zur Laufzeit. Dabei werden Teile der Formulardefinition „on the fly“ erzeugt und im Moment der Darstellung erst integriert.

Einfügen eines Request aus einem Servlet

Diese Beispiel zeigt den Zugriff auf ein Servlet zur Laufzeit. Dabei werden die Ausgabedaten der Servlet-Antwort vor der Integration in das Framework mittels XSLT in eine passende Form gebracht. In Beispiel wird eine MyCoRe-Klassifikation in eine Auswahlliste eingefügt.

```
<list id="COrigin" width="300" type="dropdown">
 <item value=""><label xml:lang="de">(bitte wählen)</label></item>
 <include uri="request:servlets/MCRQueryServlet?XSL.Style=classif-to-items&am
p;type=class&amp;hosts=local&amp;lang=de&amp;query=/mycoreclass%
5b@ID='DocPortal_class_00000002'%5d" />
</list>
```

Abbildung 3.39: Include Servlet-Request

Einfügen eines Definitionsabschnittes aus der Session

Es besteht auch die Möglichkeit, Daten dynamisch in der MCRSession zu hinterlegen und von dort in die Gestaltung des Formulars zu integrieren. Im ersten Schritt wird in einer Java-Klasse ein JDOM-Element erzeugt, welches dann im zweiten Schritt in das Formular eingebaut wird. Das Attribut **cacheable** spezifiziert dabei, ob die Daten permanent vorgehalten werden sollen. Für ständig wechselnde daten ist hier **false** anzugeben.

```
org.jdom.Element items = new org.jdom.Element(„items“);
org.jdom.Element item = new org.jdom.Element(„item“);
item.setAttribute(...)
items.addContent(item);
...
MCRSessionMgr.getCurrentSession().put(„mylist“,items);
```

Abbildung 3.40: Java-Code

```
<list id="CSelect" width="300" type="dropdown">
 <include uri=“session:mylist“ cacheable=“false“ />
</list>
```

Abbildung 3.41: Include Session-Daten

4.Anmerkungen und Hinweise