

# MyCoRe Progammer Guide

Frank Lützenkirchen (Essen/Duisburg)  
Jens Kupferschmidt (Leipzig)  
Detlev Degenhardt (Freiburg)  
Johannes Bühler (Greifswald)  
Heiko Helmbrecht (München)  
Thomas Scheffler (Jena)

**Release 1.2**

4. November 2005

## Inhaltsverzeichnis

1. Softwareentwicklung.....	8
1.1 Vorabbemerkungen.....	8
1.2 CVS-Zugang.....	8
2. Allgemeines der Implementierungen.....	9
2.1 Klassen, Pakete und Verantwortlichkeiten.....	9
2.2 Allgemeine Klassen / Exception-Modell / MCRCache.....	9
2.3 Das Datenmodell-API-Konzept allgemein.....	9
2.4 Die Session-Verwaltung.....	9
2.5 Das EventHandler-Modell.....	11
2.5.1 Das EventHandler-Modell am Beispiel der Metadaten-Objekte.....	11
2.5.2 Die Konfiguration des EventHandler-Managers.....	12
2.6 Das Vererbungsmodell.....	13
3. Funktionsprinzipien und Implementierungen von Kernkomponenten.....	16
3.1 Das Query-Modell von MyCoRe.....	16
3.1.1 Operatoren.....	16
3.1.2 Pfadangaben.....	17
3.1.3 Abfragen von Objekt-Metadaten.....	18
3.1.4 Das Resultat der Query.....	18
3.1.5 Abfragen von Derivaten.....	19
3.1.6 Abfragen von Klassifikationen.....	19
3.2 Die Benutzerverwaltung.....	19
3.2.1 Die Geschäftsprozesse der MyCoRe Benutzerverwaltung.....	19
3.2.2 Benutzer, Gruppen, Privilegien und Regeln.....	20
3.3 Die Backend-Stores.....	21
3.3.1 Hibernate oder nativ SQL?.....	21
3.3.2 Das Search-Backend JDOM-Tree.....	22
3.3.3 Das Search-Backend für IBM Content Manager 8.2.....	22
3.3.4 Das Search-Backend für XML:DB.....	23
3.4 Die Frontend Komponenten.....	23
3.4.1 Erweiterung des Commandline Tools.....	23
3.4.2 Das Zusammenspiel der Servlets mit dem MCRServlet.....	25
3.4.3 Das Login-Servlet und MCRSession.....	27
3.4.4 Generieren von Zip-Dateien.....	28
3.5 XML Funktionalität.....	29
3.5.1 URI Resolver.....	29
3.6 Das MyCoRe Editor Framework.....	30
3.6.1 Funktionalität.....	30
3.6.2 Architektur.....	31
3.6.3 Workarounds.....	32
3.6.4 Beschreibung der Editor-Formular-Definition.....	32
3.6.4.1 Die Grundstruktur.....	33
3.6.4.2 Abbildung zwischen Eingabefeldern und XML-Strukturen.....	37
3.6.4.3 Integration und Aufruf des Editor Framework.....	39
3.6.4.4 Ausgabeziele.....	40
3.6.5 Syntax der Formularelemente.....	42
3.6.5.1 Aufbau einer Zelle.....	42
3.6.5.2 Texteingabefelder.....	43
3.6.5.3 Passwort-Eingabefelder.....	43
3.6.5.4 Einfache Checkboxes.....	44

---

3.6.5.5 Auswahllisten.....	44
3.6.5.6 Radio-Buttons und Checkbox-Listen.....	45
3.6.5.7 Beschriftungen.....	46
3.6.5.8 Abstandshalter.....	46
3.6.5.9 Externes Popup-Fenster.....	46
3.6.5.10 Buttons.....	47
3.6.5.11 SubmitButton.....	47
3.6.5.12 CancelButton.....	48
3.6.5.13 Ausgabe von Werten aus dem Quelldokument.....	48
3.6.5.14 Wiederholbare Element mit Repeatern erstellen.....	49
3.6.5.15 Der Framework-interne FileUpload.....	49
3.6.5.16 Integration externer Datenquellen.....	51
3.6.6 Eingabevalidierung.....	52
3.6.6.1 Einführung.....	52
3.6.6.2 Validierungsregeln für einzelne Felder.....	54
3.6.6.3 Validierungsregeln für Feldkombinationen.....	55
3.6.6.4 Geplante Validierungsmöglichkeiten.....	56
4.Module.....	57
4.1 Das SimpleWorkflow-Modul.....	57
4.1.1 Allgemeines.....	57
4.1.2 Komponenten und Funktionen.....	58
4.1.3 Installation.....	59
4.1.4 Konfiguration.....	60
5.Anmerkungen und Hinweise.....	63
5.1 Ergänzung der DocPortal-Beispieldaten.....	63
5.1.1 Ergänzungen in einer Beispielgruppe.....	63
5.1.2 Hinzufügen einer neuen Beispielgruppe.....	63

## Abbildungsverzeichnis

Abbildung 2.1: Die Klassen der Sessionverwaltung.....	10
Abbildung 2.2: Klassendiagramm des EventHandler-Modells.....	12
Abbildung 2.3: Auszug aus dem Metadaten-Objektes des Elternsatzes.....	14
Abbildung 2.4: Auszug aus dem Metadaten-Objektes des Kindsatzes.....	14
Abbildung 2.5: XML-Syntax eines Kind-Datensatzes als Query-Resultat.....	15
Abbildung 3.1: XML Syntax des Query-Resultates.....	18
Abbildung 3.2: Geschäftsprozesse der Benutzerverwaltung in MyCoRe.....	20
Abbildung 3.3: Zusammenhang der Java-Klassen.....	24
Abbildung 3.4: Klassendiagramm Common Servlets.....	25
Abbildung 3.5: Sequenzdiagramm Common Servlets.....	26
Abbildung 3.6: Ablaufdiagramm für MCRServlet.doGetPost().....	27
Abbildung 3.7: XML Output des LoginServlets.....	28
Abbildung 3.8: Ablaufdiagramm für MCRLoginServlet.doGetPost().....	28
Abbildung 3.9: : Einbindung des Editors in eine Webseite.....	31
Abbildung 3.10: Rahmen der Formular-Definition.....	33
Abbildung 3.11: Definition des Root-Panels.....	33
Abbildung 3.12: Definition eines einfachen Formulares.....	34
Abbildung 3.13: Definition der Ausrichtung und des Submit-Button.....	35
Abbildung 3.14: Auslagen von Definitionen aus dem Root-Panel.....	36
Abbildung 3.15: Auslagen von Definitionen aus dem dem Editor-Formular.....	37
Abbildung 3.16: Die imports-Formular-Definition.....	37
Abbildung 3.17: Einfügen des XML-Main-Tag-Namen.....	38
Abbildung 3.18: Integration von einfachen XML-Tags.....	38
Abbildung 3.19: Integration von mehrfachen XML-Tags.....	39
Abbildung 3.20: : Einbindung des Editors in eine Webseite.....	39
Abbildung 3.21: Codesequenz zum Aufruf des Editor Framework.....	39
Abbildung 3.22: Rahmen der Formular-Definition.....	41
Abbildung 3.23: Java-Code-Sequenz für den Zugriff.....	41
Abbildung 3.24: Rahmen der Formular-Definition mit name=value.....	42
Abbildung 3.25: Syntax des cell-Elements.....	42
Abbildung 3.26: Syntax von textfield und textarea.....	43
Abbildung 3.27: AutoFill-Werte in textfield und textarea.....	43
Abbildung 3.28: Syntax des password-Elements.....	44
Abbildung 3.29: Syntax einer Auswahlliste.....	44
Abbildung 3.30: Beispiel für Mehrfachauswahl.....	45
Abbildung 3.31: Syntax von Radio-Button und Checkbox-Listen.....	45
Abbildung 3.32: Syntax eines Textfeldes.....	46
Abbildung 3.33: Syntax eines Abstandhalters.....	46
Abbildung 3.34: Syntax eines Popup-Fensters.....	47
Abbildung 3.35: Syntax eines einfachen Buttons.....	47
Abbildung 3.36: Syntax des Submit-Buttons.....	48
Abbildung 3.37: Syntax des Cancel-Buttons.....	48
Abbildung 3.38: Beispiel Repeater.....	49
Abbildung 3.39: Syntax des FileUpload.....	50
Abbildung 3.40: Java-Code zum Lesen des FileUpload.....	51
Abbildung 3.41: Include Servlet-Request.....	52
Abbildung 3.42: Java-Code.....	52
Abbildung 3.43: Include Session-Daten.....	52

---

Abbildung 3.44: Fehlgeschlagene Eingabevalidierung.....	54
Abbildung 4.1: Grundübersicht des SimpleWorkflow.....	57
Abbildung 4.2: Ablaufschema im SimpleWorkflow.....	59
Abbildung 4.3: Beispiel-Web-Seite für den Aufruf eines Workflow.....	60
Abbildung 4.4: Beispiel für den Aufruf einer Aktion aus einer Web-Seite mit Daten aus dem Server. 60	
Abbildung 4.5: Abschnitt der Konfiguration für den SimpleWorkflow.....	61
Abbildung 5.1: Mindeststruktur einer Beispielgruppe.....	64

---

## Tabellenverzeichnis

Tabelle 3.1: Übersicht der MyCoRe-Backends.....	21
Tabelle 3.2: Felder mit Umlautnormalisierung im Search-Store.....	22
Tabelle 4.1: Privilegienliste für den SimpleWorkflow.....	58
Tabelle 4.2: Übersicht der SimpleWorkflow-Servlets.....	58

## **Vorwort**

Dieser Teil der Dokumentation ist vor allem für Applikationsprogrammierer gedacht. Er beschreibt die Designkriterien und ihre Umsetzung in der vorliegenden Version 1.2. 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 erweitert.

# 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 setzen:

```
CVS_RSH=ssh
CVSROOT=:ext:mcr\_username@server.mycore.de:/cvs
export CVS_RSH CVSROOT
```

Es empfiehlt 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.<sup>1</sup> 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<sup>2</sup>.

---

<sup>1</sup>[luetzenkirchen@bibl.uni-essen.de](mailto:luetzenkirchen@bibl.uni-essen.de)

<sup>2</sup><http://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](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 Datenmodell-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 Repräsentation des Request selbst anzusehen. Dazu müssen die Kontextinformationen aller dings an den Thread angebunden werden, was seit Java 1.2 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 zurück, die zum gerade ausgeführten Thread gehört. Die Klassen der Sessionverwaltung von MyCoRe sind auf Basis dieser Technologie implementiert (siehe Abbildung).

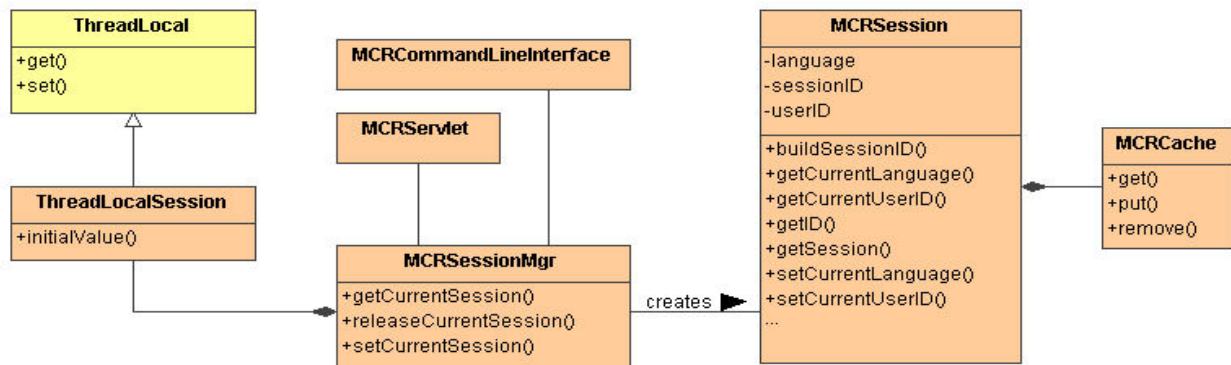


Abbildung 2.1: Die Klassen der Sessionverwaltung

Klienten der Sessionverwaltung sind alle Klassen, die Kontextinformationen lesen oder modifizieren wollen, wie zum Beispiel `MCRServlet` und `MCRCommandLineInterface`. Kontextinformationen werden als Instanzen der Klasse `MCRSession` 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 Klasse `ThreadLocalSession` 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 von `MCR Session` zurückgegeben 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 möglich 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 EventHandler-Modell

Mit Version 1.2 wurde in die MyCoRe-Implementierung ein EventHandler-Basispaket integriert. Ziel ist es, eine bessere Trennung der Code-Schichten des Datenmodells und der Backends zu erreichen. Im Datenmodell sollen nur noch Ereignisse ausgelöst werden (z. B. create, delete usw.), welche dann bestimmt durch die Konfiguration in den Property-Dateien verarbeitet werden. Es soll ein allgemeingültiges Template-Modell existieren, welches für die erforderlichen Anwendungsfälle ausgebaut werden kann. Ein singleton-Manager-Prozess nimmt nur ein Ereignis entgegen, wählt die dafür bestimmte Konfiguration aus und startet die Methode **doHandleEvent(MCRevent evt)**. Dies geschieht in der Reihenfolge, welche in der Konfiguration angegeben ist und stellt ein Pipeline-Verfahren dar. Das Event-Object wird dabei nacheinander an die Handler durchgereicht. Änderungen an den im Event-Object gespeicherten Daten werden also für alle folgenden Handler wirksam. Kommt es bei einem Handler zu einer Ausnahme, so wird diese vom Manager aufgefangen und es wird für alle in der Pipeline davor liegenden Handler die Methode **undoHandleEvent(MCRevent evt)** initiiert. Somit ist ein Rollback möglich. Je nach Anwendung ist es möglich, verschiedene Pipelines für unterschiedliche Abläufe unabhängig voneinander zu implementieren, z. B. eine Pipeline für die Verarbeitung der Metadaten und eine andere für die Volltextindizierung der Dokumente. Die Pipelines und die Damit verbundenen Ereignisse unterscheiden sich am Namen der jeweiligen Pipeline.

### 2.5.1 Das EventHandler-Modell am Beispiel der Metadaten-Objekte

Das EventHandler-Modell wird beispielsweise eingesetzt, um Objekte vom Typ MCRObjekt persistent zu speichern. Das nachfolgenden Klassendiagramm soll die Zusammenhänge verdeutlichen.

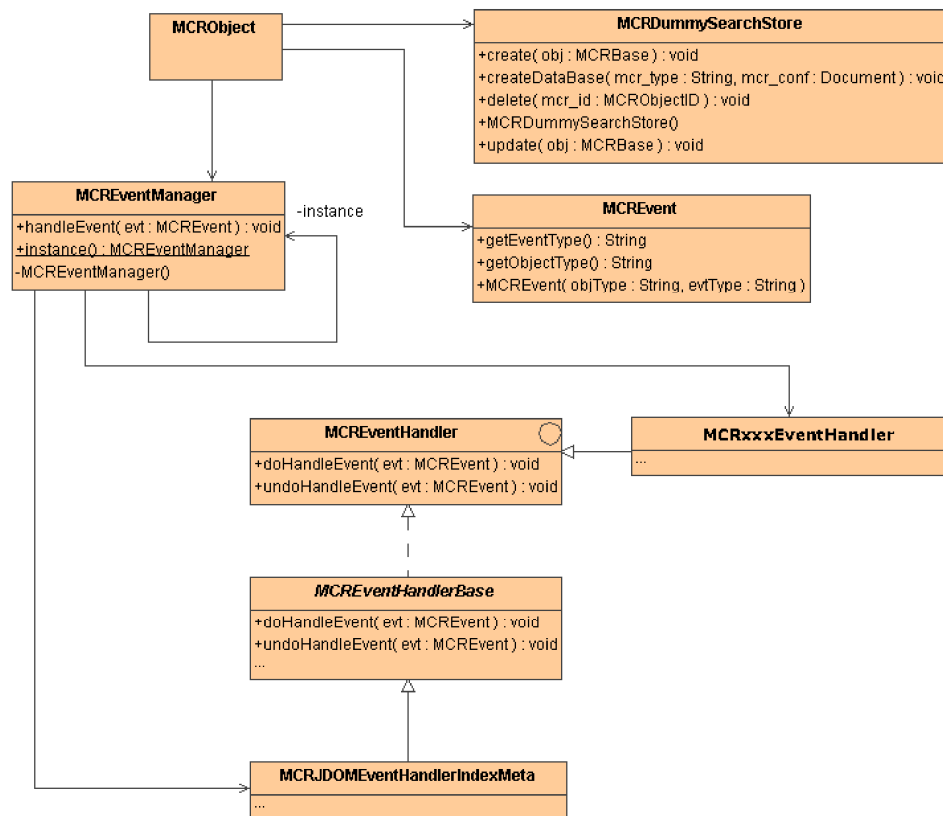


Abbildung 2.2 Klassendiagramm des EventHandler-Modells

1. Das **MCRObject** ruft in MyCoRe-Version 1.2 zuerst eine Persistence-Layer-Implementierung nach alter Konzeption auf. Hier wurde zur Nutzung des EventHandlers eine Dummy-Klasse **MCRDummySearchStore** geschaffen, welche keine Funktionalität ausführt.
2. Anschließend wird von **MCRObject** ein neues Ereignis erzeugt, welches in diesem Fall die vordefinierte Pipeline `OBJECT_TYPE` und das vordefinierte Ereignis `CREATE_EVENT` nutzt. Es können aber auch beliebige Strings eingetragen werden. dabei ist aber auf die Konsistenz zu achten.

```
MCREvent evt = new MCREvent(MCREvent.OBJECT_TYPE, MCREvent.CREATE_EVENT);
```

3. Nun wird dem neuen Ereignis das Datum übergeben, welches an die Handler weiter gereicht werden soll. Ein Ereignis kann auch mehrere Daten beinhalten.  

```
evt.put("object", this);
```
4. Die folgende Zeile ruft abschließend den **MCREventManager** auf und stößt die Handler für die Pipeline an.  

```
MCREventManager.instance().handleEvent(evt);
```

### 2.5.2 Die Konfiguration des EventHandler-Managers

Alle Konfigurationen befinden sich im Verzeichnis der Applikation (z. B. DocPortal) in der Datei *mycore.properties.private* (bzw. *mycore.properties.private.template*).

In der Version 1.2 von MyCoRe ist es noch erforderlich, für den jeweiligen SearchStore die dummy-Klasse anzugeben:

```
MCR.persistence_jdom_class_name=org.mycore.common.events.MCRDummySearchStore
```

Nun müssen noch die EventHandler für jede Pipeline (in diesem Fall ist es MCRObjekt = MCREvent.OBJECT\_TYPE) in der Reihenfolge ihrer Ausführung angegeben werden. Jeder Handler bekommt dabei eine aufsteigende Nummer.

```
MCR.EventHandler.MCRObjekt.1.class=org.mycore.backend.jdom.MCRJDOMEEventHandlerIndexMeta
```

Wollen Sie eigene EventHandler schreiben und diese Einbinden, so ist es ratsam diese direkt von MCREventHandler abzuleiten und analog zu den bestehenden Handlern einzubinden. Sie können dafür auch frei neue Pipelines und Ereignisse definieren. Den MCREventManager können Sie nun an beliebiger Code-Stelle einbauen und ihm ein von Ihnen definiertes Ereignis übergeben. Diese Komponente ist allgemein verwendbar und nicht auf das MyCoRe-Datenmodell festgelegt.

## 2.6 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 verschiedenen 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">Kapitel 1</title>
</titles>
<sizes class="MCRMetalangText" heritable="true" notinherit="true"..>
<size xml:lang="de">14 Seiten</size>
</sizes>
...
</metadata>
....
</mycoreobject>

```

Abbildung 2.4: Auszug aus dem Metadaten-Objektes des Kindsatzes

```

....
</mycoreobject>

```

Abbildung 2.3: Auszug aus dem Metadaten-Objektes des Elternsatzes

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.<sup>3</sup>

<sup>3</sup> Siehe XML-Syntax im User Guide.

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.

```
<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 2.5: XML-Syntax eines Kind-Datensatzes als Query-Resultat

## 3. Funktionsprinzipien und Implementierungen von Kernkomponenten

### 3.1 Das Query-Modell 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 Spezifikationen 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 kleinen 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<sup>4</sup> 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 Persistence-Layern. Wenn weitere Operatoren benötigt werden, so müssen diese in **ALLEN** Persistence-Implementierungen eingebaut werden.

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

- Operator = - ist für alle Values zulässig

---

<sup>4</sup> 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 eine 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, je nachdem, 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 Persistence-Layern nicht erlaubt (und in der Praxis auch nicht erforderlich).

```

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

```

Hier noch einige Hinweise:

- \* als SpecialNodeFunction darf allein nur in einer Single Query ohne OutPath angegeben werden. Es erfolgt dann die Suche über alle für TextSearch markierte Metadaten.
- **text()** als SpecialNodeFunction darf allein nur in einer Single Query ohne OutPath angegeben werden. Es wird nach 'ts()' überführt.
- **doctext()** als SpecialNodeFunction ist für die Abfrage des TextSearch des Dokument-Objektes vorgesehen.
- In MyCoRe kann bei Bedarf der der Applikation um weitere SpecialNodeFunction 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<sup>5</sup>. 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[SpecialNodeFunction]

```

Dies bedeutet, es können mehrere Abfragen hintereinander mit AND oder OR verknüpft werden. Für jedes Metadatum des Datenmodells ist eine OneQuery zu formulieren. Da 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-Container zusammengefasst und der Anwendung zurückgegeben. Der Aufbau des Containers ist dabei Persistence-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 entweder **'local'** oder den Hostalias des Servers, von dem das Resultat stammt.

<sup>5</sup> Die Typen class und derivate sind reservierte Typen.

- Das Attribut **id** ist die entsprechende MCRObjektID 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. Beachten Sie jedoch, dass das Datenmodell fest vorgeschrieben 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[SpecialNodeFunction]
```

### 3.1.6 Abfragen von Klassifikationen

Klassifikationen werden in MyCoRe-Anwendungen erfahrungsgemäß am häufigsten abgefragt. Die Klassifikationen können nur nach einem sehr festen Schema abgefragt werden, wobei entweder die gesamte Klassifikation oder nur eine einzelne Kategorie zurückgegeben wird. Die XML-Struktur entspricht dabei immer der einer Klassifikation.

```
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 (siehe Abbildung) 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 definieren).

Offensichtlich dürfen nicht alle Akteure des Systems die Berechtigung haben, alle Geschäftsprozesse durchzuführen zu können. Daher muss 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 Regeln 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.

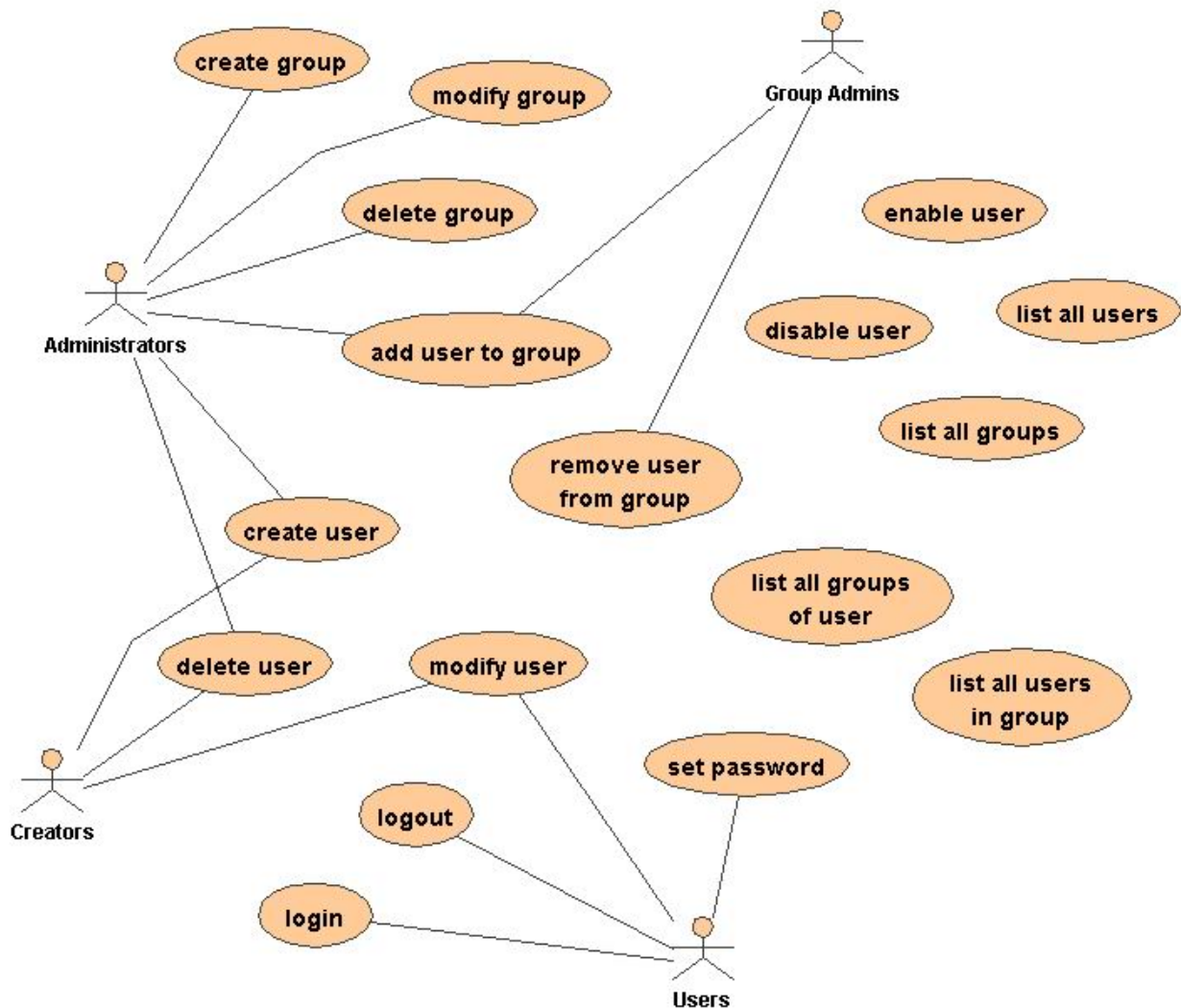


Abbildung 3.2: Geschäftsprozesse der Benutzerverwaltung in MyCoRe

### 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 Die Backend-Stores

Im Backend-Bereich muss zwischen den verschiedenen Aufgaben der Stores unterschieden werden. Es gib Stores für die Speicherung organisatorischer Informationen (z. B. User-Daten, XML-Daten usw.), Stores für die Suche und für die Ablage der eigentlichen digitalen Objekte und deren Volltexte. Dabei kann die genutzte Backend-Software ggf. auch für mehrere Stores verwendet werden. Eine Übersicht gibt die folgende Tabelle.

Backend	Organisation	Suche	Ablage	Volltexte	Bemerkung
HSQLDB	x				frei verfügbar; für kleine Lösungen
MySQL	x				frei verfügbar; für mittlere Lösungen
IBM DB2	x				kommerzielles Produkt; für große Lösungen
JDOM-Tree		x			in MyCore included; für kleine Lösungen
nativ SQL		x			benötigt ein SQL / Hibernate-Backend; je nach Backend
Lucene		x		x	frei verfügbar; für mittlere bis große Lösungen
eXist		x			frei verfügbar; für mittlere Lösungen
IBM CM 8.x		x	x	x	kommerzielles Produkt, für große Lösungen
FileSystem			x		je nach Plattenplatz
Helix			x		kommerzielles Produkt; für Audio-/Video-Lösungen

Tabelle 3.1: Übersicht der MyCoRe-Backends

#### 3.3.1 Hibernate oder nativ SQL?

MyCoRe in der Version 1.2 bietet 2 Möglichkeiten, die Stores für die organisatorischen Daten einzubinden.

Zum einen wird über das Package `org.mycore.backend.sql` ein direkter Zugriff auf relationale Datenbanken via JDBC realisiert. Der Vorteil davon sind optimale Zugriffszeiten. Nachteilig kann sich auswirken, dass wir nur einen Bruchteil der am Markt verfügbaren Datenbanken integriert und getestet haben. Es kann also bei Verwendung anderer Datenbanken ggf. zu Problemen kommen (besonders beim automatischen Anlegen der Tabellen), da diese nicht von uns getestet wurden.

Der zweite Weg ist die Nutzung unserer Hibernate-Integration. Hier übernimmt das freie Paket Hibernate die Anpassung an die jeweils darunter liegende Datenbank. Es wird also der gesamte Zugriff über ein fest definiertes API geregelt. Der Nachteil ist ein leichter Performance-Verlust, da ja alle Daten durch das API verwaltet werden. Die Klassen zur Arbeit mit Hibernate stehen in `org.mycore.backend.hibernate`.

Welche der beiden Zugriffsarten nun in Ihrem konkreten Projekt genommen wird hängt von den ganz spezifischen Eigenschaften der Anwendung und deren Umgebung sowie den personellen Ressourcen ab. Das MyCoRe-System wurde mit beiden Varianten getestet.

Hinweis:

Während es in der nativ-SQL-Anwendung möglich ist, für jeden XML-Daten-Store einen eigene Tabelle anzugeben, werden die XML-Daten unter Hibernate immer in einer gemeinsamen Tabelle gehalten (Property MCR.xml\_store\_sql\_table).

### 3.3.2 Das Search-Backend JDOM-Tree

Das MyCoRe-Paket bietet eine simple Standard-Lösung für die Suche in kleinen Beispielanwendungen, ohne das zusätzliche externe Produkte verwendet werden müssen. Die XML-Daten werden im JDOM-Tree zum Startzeitpunkt der Applikation direkt aus der XML-SQL-Tabelle gelesen, teilweise hinsichtlich der Umlaute normalisiert und im weiteren Verlauf der Anwendung im Hauptspeicher verwaltet. Create / Update / Delete auf die XML-SQL-Tabellen wird direkt mit dem im Speicher befindlichen Datenbaum synchronisiert.

Die Xpath-Suchanfrage wird in ein XSLT-Stylesheet umgewandelt, welches gegen die XML-daten im Hauptspeicher läuft. Als Ergebnis wird die Liste der Treffer-ID's zurückgegeben. Für die Datumssuche wurde eine zusätzliche XSLT-Funktion implementiert. Suchdaten im Operator contains werden hinsichtlich der Umlaute normalisiert (z. B. contains(„Eindrücke“) --> contains („eindruecke“)).

Metadaten-Typ	Felder
MCRMetaLangText	tag/subtag/text()
MCRMetaPersonName	tag/subtag/firstname/text() tag/subtag/callname/text() tag/subtag/surname/text() tag/subtag/fullname/text()
MCRMetaInstitutionName	tag/subtag/fullname/text()

Tabelle 3.2: Felder mit Umlautnormalisierung im Search-Store

Hinweis:

Dieser Search-Store ist nur bis zu einigen 100 Datensätzen performant und nicht für Produktionssysteme gedacht.

### 3.3.3 Das Search-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\***.

### 3.3.4 Das Search-Backend für XML:DB

Muf dem Markt gibt es ja eine Reihe von XML:DB's. Im Bereich der kommerziellen Systeme wurde für MyCore einmal prototypisch Tamino getestet. Aus Kostengründen wurde jedoch auf eine Produktionsumgebung verzichtet. Im Bereich freier Software kommt bei MyCoRe das Produkt eXist ins Spiel. Für Projekte, die mittlere Leistungsanforderungen haben (bis ca. 8000 Datensätze) ist eXist auch im Produktionsbetrieb im Einsatz. Bei größeren datenmengen kann es derzeit zu Performance-Problemem im Bereich Update kommen.

eXist ist extern zu beziehen und zu installieren. Anschließend wird es über die Konfiguration eingebunden (siehe UserGuide). Analog zum JDOM-Backend wird auch für das XML:DB Backend eine Umlautnormalisierung für die durchsuchbaren Daten und die Queries vorgenommen. Als Ergebnis der Suche wird die Liste der Treffer-ID's zurückgegeben.

Hinweis:

eXist ignoriert derzeit einige Schachtelungskonstrukte einer Query 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. In DocPortal 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.

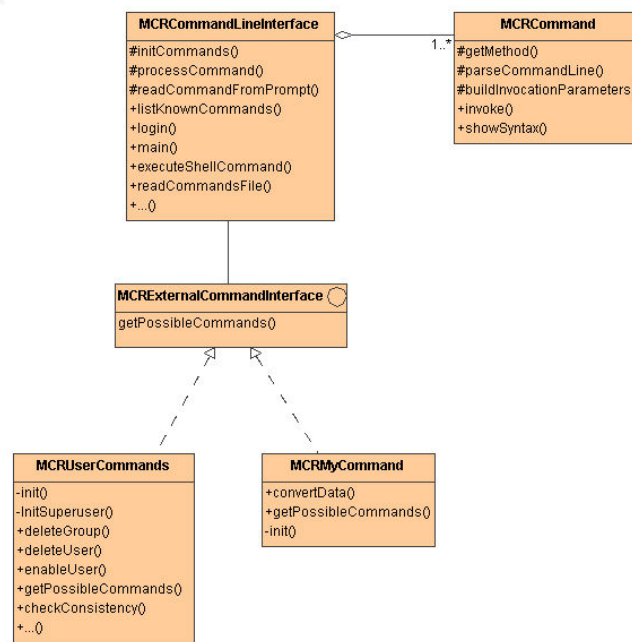


Abbildung 3.3: Zusammenhang der Java-Klassen

Es ist relativ einfach, weitere Kommandos hinzuzufügen. In DocPortal 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-Klasse z. B. `MCRTestCommand.java`. Die Klasse **kann** im Package **org.mycore.frontend.cli** liegen, sie können Sie Aber auch in den Bereich tun, zu dem es logisch gehört.
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.<sup>6</sup> 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.<sup>7</sup>
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.

<sup>6</sup> siehe Beispielcode

<sup>7</sup> 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.

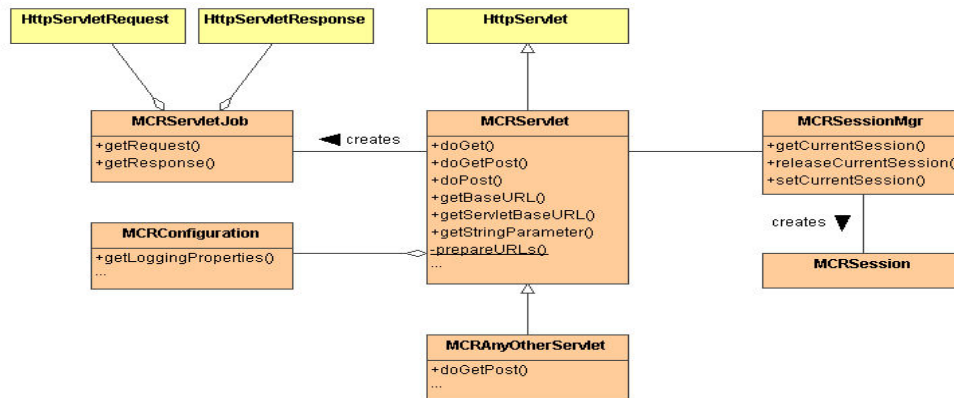


Abbildung 3.4: Klassendiagramm Common Servlets

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 in einem anderen Kapitel beschrieben.

MCRServlet selbst ist direkt von HttpServlet abgeleitet. Sollen andere Servlets im MoCoRe-Softwaresystem die von MCRServlet angebotenen Funktionen automatisch nutzen, so müssen 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 (siehe Abbildung) dargestellt. Bei einem eingehenden Request (doGet() oder doPost()) wird zunächst an MCRServlet.doGetPost() delegiert.<sup>8</sup>

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

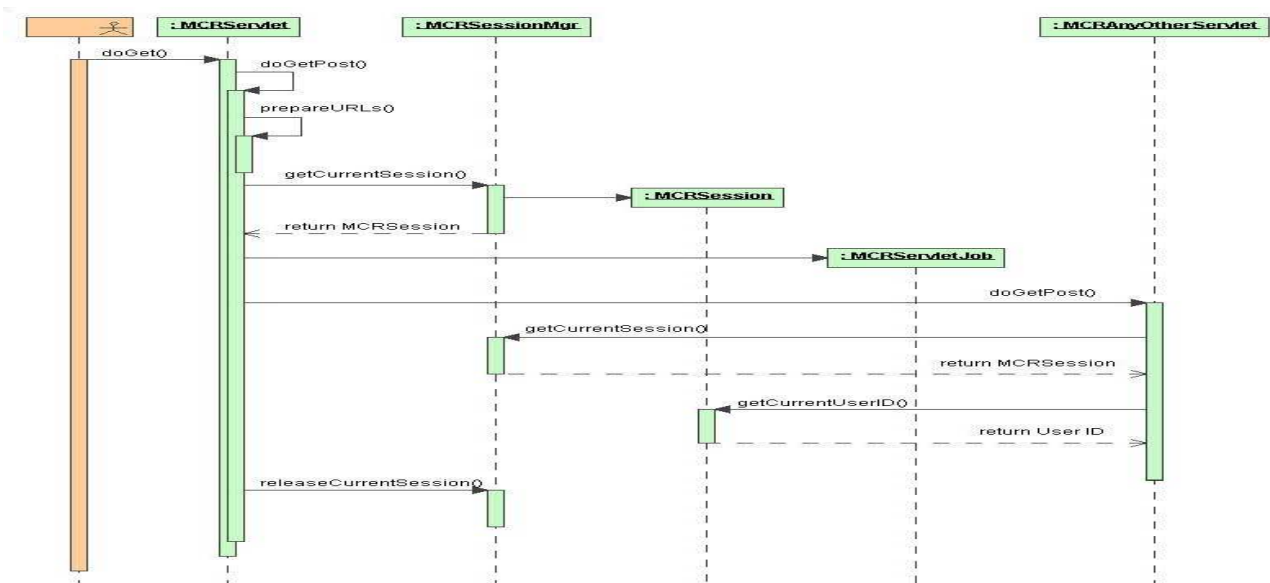
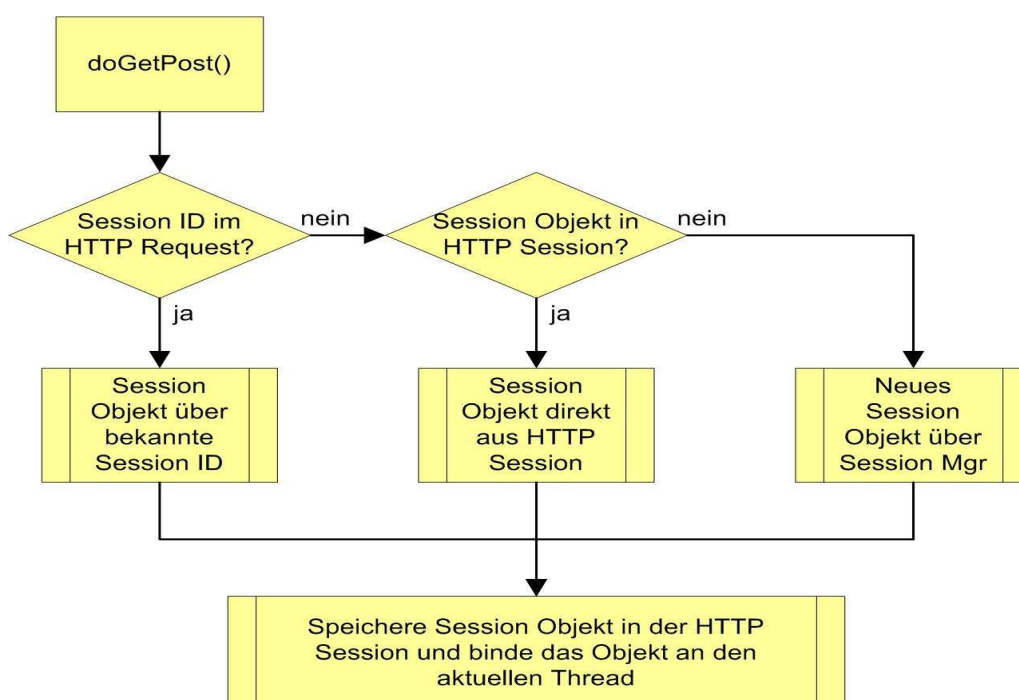


Abbildung 3.5: Sequenzdiagramm Common Servlets

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, so wird ein neues Session-Objekt über den Aufruf von `MCRSessionMgr.getCurrentSession()` erzeugt. Nachfolgend wird das Session-Objekt an den aktuellen Thread gebunden und zusätzlich in der `HttpSession` abgelegt.



**Abbildung 3.6:** Ablaufdiagramm für `MCRServlet.doGetPost()`

Im Sequenzdiagramm gehen wir davon aus, 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).<sup>9</sup>

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 Kontextinformationen 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 zunächst die Funktionalität von `MCRServlet` ausgenutzt und die in Abschnitt 3.7.2 beschriebene Verbindung zur Sessionverwaltung hergestellt. Wie dort ebenfalls beschrieben, wird der Programmfluss an das Login-Servlet über die Methode `MCRLoginServlet.doGetPost()` delegiert. Der Ablauf in `doGetPost()` wird im folgenden Diagramm dargestellt und ist selbsterklärend.

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

---

<sup>9</sup> 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 zugewiesen 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

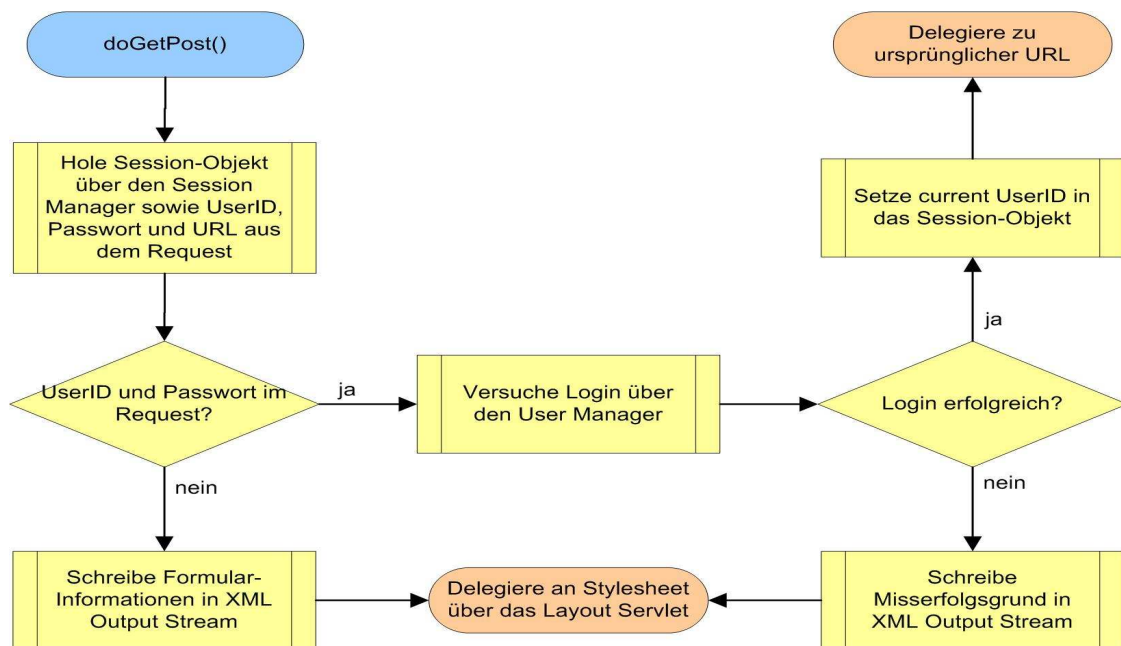


Abbildung 3.8: Ablaufdiagramm für MCRLoginServlet.doGetPost()

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 Metadaten transformation in das gewünschte Auslieferungsformat vornimmt. In DocPortal 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.MCRURIRResolver` 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 Schlüssel 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.

#### **mrobject:[MCRObjctID]**

liest die XML-Darstellung der Metadaten eines MCRObjct, z. B.

mrobject:DocPortal\_document\_07910401

#### **query:[Query]**

liest ein XML Element, das als Wurzelement eines Ergebnisses der Suchanfrage zurück geliefert wird. [Query] ist dabei ähnlich dem Query-Parametern in URLs aufgebaut, d. h. dass Parameter durch ein „&“ getrennt werden (maskiert als „&amp;“) und ein Name-Wert-Paar darstellen. Der Name ist vom Wert durch ein „=“-Zeichen getrennt. Namen und Werte müssen URL-encodiert vorliegen, der Zeichensatz ergibt sich aus der Property „MCR.request\_charencoding“. Folgende Parameter werden ausgewertet.

- host: Eine kommaseparierte Liste von Host-Aliasen, an die Anfrage gestellt wird. Dieser Parameter ist optional.
- type: Der Dokumenttyp über den gesucht wird.
- query: eine gültige MyCoRe-Query (siehe Kapitel 3.1)

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

file http https request resource session webapp mrobject

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

file http https resource session query webapp mrobject

## **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**)<sup>10</sup> 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*)<sup>11</sup> mittels eines dazu passenden XSL Stylesheets (*anypage.xsl*)<sup>12</sup> 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

<sup>10</sup> In den MyCoRe-Applikationen meint eine Mischung aus dem MCRObjektID-Typ und der Bezeichnung eines Verarbeitungsschrittes (editor-commit-document.xml).

<sup>11</sup> z. B. editor\_form\_...xml in DocPortal

<sup>12</sup> z. B. MyCoReWePage.xsl in DocPortal

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 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 Formulars

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 zugeordnet. 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 angegebenen 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 MCRLayServlet 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 ermöglichen 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 Submit-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 Aufbau 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** beschrieben. **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 einzeliges 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 Attribut **height** gibt für mehrzeilige Texteingabefelder die Anzahl Zeilen an. Das Attribut **maxlength** 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: Syntax des password-Elements

### 3.6.5.4 Einfache Checkboxes

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.

```
<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>
```

Abbildung 3.30: Beispiel für Mehrfachauswahl

Das Beispiel 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 Anordnung 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 3.31: 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 realisiert 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 3.32: 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 3.33: Syntax eines Abstandhalters

### 3.6.5.9 Externes Popup-Fenster

Ein externes Popup-Fenster kann z. B. ein Hilfetext zu einem Eingabefeld im HTML-Format enthalten. Es können aber auch beliebige andere Informationen und Funktionalitäten wie Eingabehilfen dargestellt werden. Der Startpunkt ist ein Button im Formular, der mit einer frei wählbaren Beschriftung versehen werden kann. Standardmäßig wird er als [?] Button im Formular dargestellt. Bei Anklicken des Buttons erscheint das codierte HTML in einem Popup-Fenster. Die Attribute **width** und **height** steuern die Breite und Höhe dieses Fensters. Das Attribut **css** gestattet die Verwendung eines externen CSS-Files. Wichtig ist, dass jedes **helpPopup** Element eine eindeutige ID besitzt. Optional kann der Popup-Fenster 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.

Innerhalb des **helpPopup**-Tags gibt es mehrere Tags, welche das Aussehen und den Inhalt des Aufruf-Buttons und des Popup-Fensters bestimmen. Jedes Tag verfügt über das **xml:lang**-Attribut.

Somit kann jedes Tag für die im Projekt benötigten Sprachen implementiert werden. Zusätzlich gibt es den Konstrukt **xml:lang="all"**, der anzeigt, dass das entsprechende Tag immer unabhängig von der Spracheinstellung des Systems präsentiert werden soll.

Im Einzelnen sind die Tags wie folgend definiert:

- **button** – Gibt die Zeichenkette an, welche der Start-Button haben soll.
- **title** – Gibt den Titel des Popup-Fensters an.
- **close** – Gibt den Text zum Schließen des Popup-Fensters an.
- **label** – Beinhaltet den eigentlichen darzustellenden HTML-Code oder Text.

```
<helpPopup id="url.help" width="400px" height="200px" css="/mycss.css">
  <button xml:lang="all">!</button>
  <title xml:lang="de">Hilfetext</title>
  <close xml:lang="de">Beenden</close>
  <label xml:lang="de">
    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.
  </label>
</helpPopup>

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

Abbildung 3.34: Syntax eines Popup-Fensters

### 3.6.5.10 Buttons

Über das **button** Element wird ein Knopf erzeugt, der bei Anklicken zu einer externen URL wechselt. Das Attribut **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 3.35: 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 Attribut **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 3.36: 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.<sup>13</sup> Das Attribut **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 3.37: 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>
```

<sup>13</sup> Siehe auch **Integration und Aufruf des Editor Framework**



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

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

### 3.6.5.14 Wiederholbare Element mit Repeatern erstellen

Häufig sind einzelne 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.

```
<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>
```

Abbildung 3.38: Beispiel Repeater

Das Beispiel 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. Voraussetzung ist hierfür die vollständige Integration des Upload-Service in den jeweiligen Java-Klassen.<sup>14</sup> Das

<sup>14</sup> in DocPortal ist derzeit nur der Upload integriert.

Attribut **maxlength** 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 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 Dateinhalt 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 3.39: 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

# Temporärer Speicherort für hochgeladene Dateien:

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

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

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/>).

```
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 3.40: Java-Code zum Lesen des 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&hosts=local&lang=de&query=/mycoreclass%
5b@ID='DocPortal_class_00000002'%5d" />
</list>
```

Abbildung 3.41: 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 3.42: Java-Code

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

Abbildung 3.43: Include Session-Daten

## 3.6.6 Eingabevalidierung

### 3.6.6.1 Einführung

Die Eingabefelder von Editor-Formularen können durch Integration von Validierungsregeln direkt durch das Editor-Servlet geprüft werden, noch bevor die Eingaben an das weiterverarbeitende

Zielservlet weitergegeben werden. Wenn eine Eingabe fehlerhaft ist, wird das Formular noch einmal angezeigt. Es erscheint eine Fehlermeldung. Eingabefelder mit fehlerhaften Eingaben werden im Formular optisch markiert. Zunächst ein einfaches Beispiel:

```
<editor id="validation.demo">
  <source ... />
  <target ... />

  <validationMessage>
    <label xml:lang="de">
      Eingabefehler: Bitte korrigieren Sie die markierten Felder.
    </label>
  </validationMessage>

  <components root="root" var="/document">
    ...
    <panel id="root" lines="on">
      <cell row="1" col="1" anchor="EAST">
        <text label="Titel:" />
      </cell>
      <cell row="1" col="2" anchor="WEST" var="title">
        <textfield id="tf.title" width="80">
          <condition id="cond.title" required="true">
            <label xml:lang="de">Bitte geben Sie einen Titel ein!</label>
          </condition>
        </textfield>
      </cell>
    </panel>
    ...
  </components>
</editor>
```

Das Element **validationMessage** enthält die Meldung, die am Kopf des Formulars erscheint, wenn ein oder mehrere Eingabefelder fehlerhaft ausgefüllt wurden. Dieses Element ist optional, es kann ein sprachunabhängiges label Attribut oder ein oder mehrere sprachabhängige label Elemente enthalten.

**Jedes zu validierende Eingabefeld muss eine eindeutige ID besitzen.** Neben Texteingabefeldern können auch alle anderen Feldtypen wie z. B. Auswahllisten validiert werden. Nur hidden-Felder können momentan nicht validiert werden.

Das zu validierende Feld muss ein oder mehrere **condition** Elemente enthalten, die wiederum eine **eindeutige ID** besitzen müssen. Die Attribute dieser condition Elemente enthalten die zu prüfenden Validierungsregeln. Ein condition Element kann ein sprachunabhängiges label Attribut oder ein oder mehrere sprachabhängige label Elemente enthalten, die eine Meldung für den Benutzer enthalten. Bei einer fehlerhaften Eingabe wird diese Meldung angezeigt, wenn der Benutzer mit der Maus über das Fehlersymbol fährt, das neben dem Eingabefeld angezeigt wird.

Dokumentbeschreibung bearbeiten: (Neues Dokument)

Eingabefehler: Bitte korrigieren Sie die markierten Felder.  
Wenn Sie den Mauszeiger auf das rote Symbol neben dem fehlerhaften Eingabefeld bewegen, sehen Sie weitere Hinweise.

Titel: Deutsch [!]

Person(en): Autor [Auswählen] (bitte wählen oder eingeben) [!]

Medientyp(en): Text [!]

Dokumententyp(en): (bitte wählen) [!]

Fachbereich(e): (bitte wählen) [!]

Sprache: Deutsch

Abbildung 3.44: Fehlgeschlagene Eingabevalidierung

Ein Eingabefeld kann mehr als ein condition Element enthalten. Dadurch können komplexe Validierungsregeln schrittweise geprüft werden, und der Nutzer bekommt eine exaktere Rückmeldung, welche dieser Regeln verletzt wurde. Wenn die Validierungsregeln auf mehrere condition Elemente aufgeteilt wird, sollte die „required“ Regel ggf. die erste sein.

```
<condition id="cond.title.required" required="true">
  <label>Bitte geben Sie einen Titel ein!</label>
</condition>
<condition id="cond.title.length" maxLength="250">
  <label>Bitte geben Sie maximal 250 Zeichen ein!</label>
</condition>
```

oder

```
<condition id="cond.title" required="true" maxLength="250">
  <label>Bitte geben Sie einen Titel ein (maximal 250 Zeichen)!</label>
</condition>
```

### 3.6.6.2 Validierungsregeln für einzelne Felder

Die Validierungsregeln für Eingabefelder werden über Attribute oder Attributkombinationen des condition Elements definiert. Die folgenden Regeln können geprüft werden:

#### **required="true"**

Es ist eine Eingabe erforderlich. Auch Eingaben, die nur aus Leerzeichen bestehen, werden abgewiesen. Ist diese Regel verletzt, werden ggf. weitere vorhandene Regeln nicht mehr geprüft.

#### **minLength="10"**

Die Eingabe muss aus minimal 10 Zeichen bestehen.

#### **maxLength="250"**

Die Eingabe darf aus maximal 250 Zeichen bestehen.

#### **type="integer" min="0" max="100"**

Die Eingabe muss eine ganze Zahl sein. Optional kann über die Attribute min und/oder max der Wertebereich eingeschränkt werden.

#### **type="decimal" format="de" min="0" max="3,5"**

Die Eingabe muss eine Zahl (ggf. mit Nachkommastellen) sein. Optional kann über die Attribute min und/oder max der Wertebereich eingeschränkt werden. Das Attribut format muss einen ISO 639 Sprachcode enthalten (z.B. „de“ oder „en“), der bestimmt, wie Kommazahlen formatiert werden. Die min und max Werte müssen entsprechend

diesem Format angegeben werden.

**type="datetime" format="dd.MM.yyyy" min="01.01.1970" max="31.12.2030"**

Die Eingabe muss ein Datums- oder Zeitwert sein. Optional kann über die Attribute min und/oder max der Wertebereich eingeschränkt werden. Das Attribut format muss ein java.text.SimpleDateFormat Pattern enthalten (z.B. „dd.MM.yyyy“), das bestimmt, wie der Wert formatiert sein muss. Die min und max Werte müssen entsprechend diesem Format angegeben werden.

**type="string" min="a" max="zzz"**

Ein Eingabe kann ein beliebiger Text sein. Optional kann über die Attribute min und/oder max der Wertebereich eingeschränkt werden.

**regexp=".+@.+\\.+"**

Die Eingabe muss den angegebenen regulären Ausdruck erfüllen. Die Syntax des regulären Ausdrucks ist durch die Klasse java.util.regex.Pattern definiert.

**xsl="contains(.,'@')"**

Die Eingabe wird gegen eine XSL Bedingung geprüft, wie sie in XSL if oder when Elementen verwendet werden kann. Der Eingabewert wird in der Bedingung durch einen Punkt referenziert.

### 3.6.6.3 Validierungsregeln für Feldkombinationen

Es ist ebenfalls möglich, die Eingaben zweier Felder gegeneinander zu prüfen. Dazu wird ein condition Element in das die beiden Felder enthaltende Panel eingefügt. Die Werte der beiden Felder können dann über Vergleichsoperatoren gegeneinander geprüft werden. Beispiele:

- Ein Eingabeformular zum Ändern eines Passwortes enthält zwei Passwortfelder, so dass das Passwort wiederholt eingegeben werden muss. Die Eingaben in diesen beiden Feldern müssen identisch sein.
- Ein Eingabeformular enthält zwei Eingabefelder für einen Datumsbereich. Das Datum im Feld „validFrom“ muss kleiner oder gleich dem Datum im Feld „validTo“ sein.

Hier ein Codebeispiel:

```
<textfield id="tf.date" width="10">
  <condition id="cond.date" type="datetime" format="dd.MM.yyyy">
    <label xml:lang="de">
      Bitte geben Sie das Datum im Format TT.MM.JJJJ ein,
      z. B. 01.09.2005
    </label>
  </condition>
</textfield>

<panel lines="off">

  <cell row="2" col="1" anchor="EAST">
    <text label="Gültig von:"/>
  </cell>
  <cell row="2" col="3" anchor="EAST">
    <text label="Gültig bis:"/>
  </cell>

  <cell row="2" col="2" anchor="WEST" var="validFrom" ref="tf.date" />
  <cell row="2" col="4" anchor="WEST" var="validTo" ref="tf.date" />
```

```

<condition id="cond.valid" field1="validFrom" field2="validTo"
  operator="&lt;=" type="datetime" format="dd.MM.yyyy">
  <label>Datum "Gültig von" muss &lt;= Datum "Gültig bis" sein!</label>
</condition>

</panel>

```

Zunächst werden die Werte beider Eingabefelder validFrom und validTo nach den gleichen Regeln auf die Eingabe eines korrekten Datums geprüft. Wenn mindestens eines der Felder eine Eingabe enthält (was ggf. über ein required Attribut erzwungen werden könnte), werden die Werte der Felder miteinander verglichen.

**field1="validFrom"**

Erstes zu vergleichende Feld. Syntax und Funktionsweise entsprechen dem Verhalten des Attributes var für Zellen.

**field2="validTo"**

Zweites zu vergleichende Feld. Syntax und Funktionsweise entsprechen dem Verhalten des Attributes var für Zellen.

**operator="&lt;="**

Vergleichsoperator (hier >=). Es können die Operatoren =, >, <, >=, <=, != (für ungleich) verwendet werden.

**type="datetime"**

**format="dd.MM.yyyy"**

Datentyp und Format der Eingabe, analog zu den gleichnamigen Attributen, die im vorangehenden Abschnitt bereits erläutert wurden.

### 3.6.6.4 Geplante Validierungsmöglichkeiten

Die folgenden Validierungsmöglichkeiten sind noch nicht implementiert, sollen aber kurzfristig realisiert werden:

- Validierung von hidden Feldern. Dies ist insbesondere nötig in Verbindung mit der subselect Funktion z. B. bei einer externen Auswahl einer Klassifikationskategorie, um sicherzustellen, dass mindestens eine Kategorie ausgewählt wird.
- Validierung durch externen Java-Code. Dies ist z. B. nötig, um gültige ISBNs, NBN Prüfsummen, existierende UserIDs etc. über eine externe Java-Methode prüfen zu können.



## 4.Module

Module sind eine sinnvolle Ergänzung der MyCoRe-Kern-Komponenten. Sie können optional in die Applikationen der Anwender eingebaut werden und sollen diese funktional bereichern.

### 4.1 Das SimpleWorkflow-Modul

#### 4.1.1 Allgemeines

Für die Erstellung einfacher Anwendungen, welche nur einen relativ primitiven Arbeitsablauf bedingen, was es notwendig ein einfacher Werkzeug zur Gestaltung diese Abläufe anzubieten. So entstand die Idee des SimpleWorkflow. Eigentlich handelt es sich dabei gar nicht um einen Workflow, sondern eher um eine Menge von kleinen Werkzeugen, die über HTTP-Request zu einem interaktiven Arbeitsablauf zusammengesteckt werden können. Der Begriff Workflow soll dabei die Bearbeitungsebene zwischen dem ersten Erstellen eines Objektes und dessen Ablage in den Server und die dabei vor sich gehenden Arbeitsschritte beschreiben. Physisch handelt es sich um ein Verzeichnis *workflow*, unter welchem für jeden Objekttypen Unterverzeichnisse angelegt sind und in welchem die Daten zwischengespeichert werden. **Konsultieren Sie zum besseren Verständnis auch die Beschreibung im MyCoRe-UserGuide.**

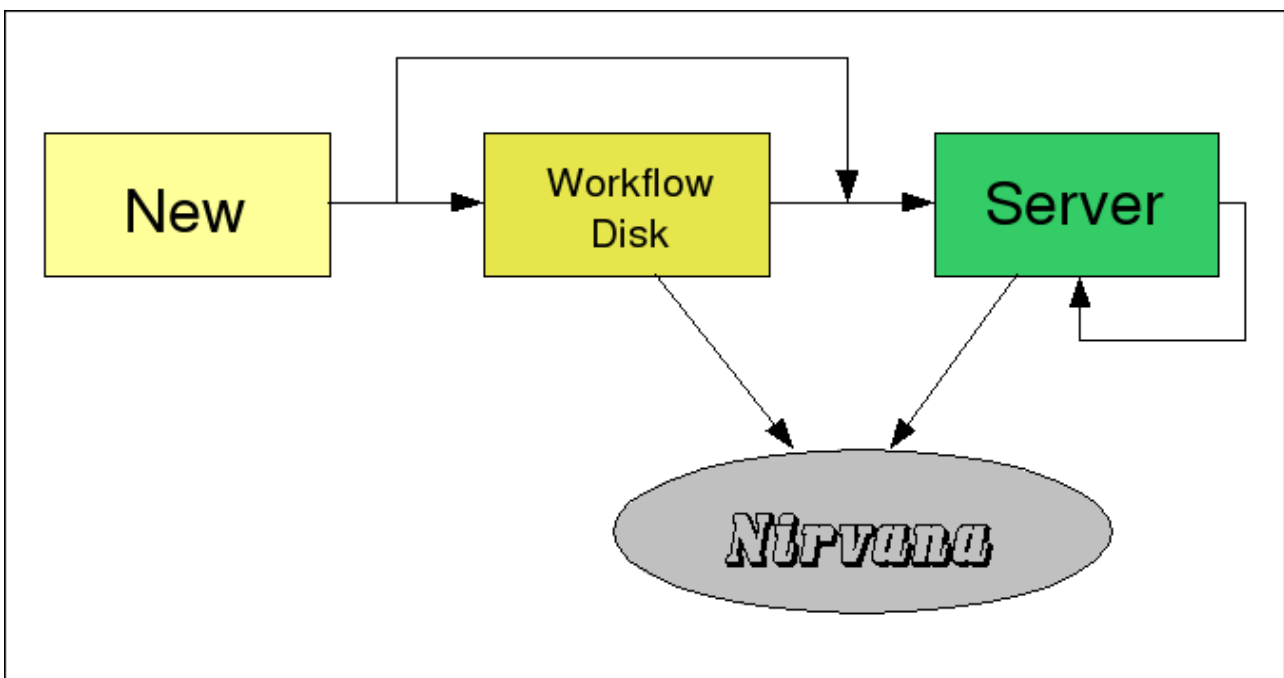


Abbildung 4.1: Grundübersicht des SimpleWorkflow

Der SimpleWorkflow besteht im wesentlichen aus einer Sammlung von Servlets, welche über HTTP-Request angesprochen, verschiedenen Bearbeitungsprozesse initiieren. Dabei wird gleichzeitig eine Berechtigungsprüfung für den Zugriff auf einzelne Aktionen gemacht. Einerseits werden dabei die allgemeinen Privilegien des aktuellen Benutzers geprüft, andererseits in den Fällen der Veränderung von Daten werden noch Eignerrechte zum Dokument mit in die Prüfung einbezogen. Die allgemeine Rechte ordnen sich wie folgend ein:

Privileg	Bedeutung
create-<type>	Gestattet das Anlegen neuer Objekte.
modify-<type>	Gestattet das Bearbeiten von Objekten im Workflow (d. h. auf dem Plattenbereich).
delete-<type>	Gestattet das Löschen von Objekten aus dem Workflow (d. h. dem Plattenbereich).
commit-<type>	Gestattet das Einstellen der Objekte aus dem Workflow (d. h. dem Plattenbereich) in den Server oder das Bearbeiten der Daten im Server.
remove-<type>	Gestattet das Löschen von Objekten aus dem Server.
editor	Dies ist ein Spezial-Privileg, um bei nicht entscheidbaren Prüfungen (z.B. durch inkorrekte Einträge) Nutzen die Bearbeitung trotzdem zu gestatten.

Tabelle 4.1: Privilegienliste für den SimpleWorkflow

### 4.1.2 Komponenten und Funktionen

Der SimpleWorkflow besteht aus einer Reihe von Servlets. Die folgende Tabelle listet die Servlets.

Servlet	Funktion
MCRStartEditorServlet	Das Servlet dient als Startpunkt für alle Arbeiten mit dem SimpleWorkflow.
MCRCheckCommitDataServlet	Wird vom Editor über ein <target>-Tag aufgerufen und schreibt die Metadaten nach Bearbeitung in den Server.
MCRCheckEditDataServlet.java	Wird vom Editor über ein <target>-Tag aufgerufen und schreibt die Metadaten nach deren Bearbeitung auf die Platte.
MCRCheckNewDataServlet.java	Wird vom Editor über ein <target>-Tag aufgerufen und schreibt die neuen Metadaten auf die Platte.
MCRCheckCommitFileServlet	Wird vom Editor über ein <target>-Tag aufgerufen und schreibt die Derivate-Daten nach Bearbeitung in den Server.
MCRCheckNewFileServlet	Wird vom Editor über ein <target>-Tag aufgerufen und schreibt die neuen Derivate-Daten auf die Platte.
MCRFileListWorkflowServlet	Listet die auf der Platte befindlichen Dateien in den Derivaten aus.
MCRFileViewWorkflowServlet	Gestattet den Zugriff auf eine Derivate-Datei auf der Platte.
MCRListDerivateServlet	Listet alle auf der Platte befindlichen Derivate aus.
MCRListWorkflowServlet	Erzeugt einen XML-Baum, welcher zur Darstellung des Workflow (Platteninhaltes) benötigt wird.

Tabelle 4.2: Übersicht der SimpleWorkflow-Servlets

Die folgende Abbildung soll noch einmal die Beziehungen der einzelnen Komponenten verdeutlichen. Hier gibt es zwei Komplexe. Der erste arbeitet mit dem Plattenzwischenspeicher und stellt einen simplen Workbasket dar. In diesen Korb können Objekte neu eingestellt, bearbeitet,

ergänzt, geprüft oder wieder gelöscht werden. Ist dieser Arbeitsschritt fertig, so kann das Objekt in den Server hoch geladen werden. Hier gibt es wieder die Möglichkeit, so der Nutzer die Berechtigung dazu hat, Objekte zu bearbeiten, verändern oder zu löschen. Alle diese Schritte arbeiten direkt gegen den Server. Ausgangspunkt aller Aktivitäten ist dabei das `MCRStartEditorServlet`. Hier wird beim Aufruf eine Aktion mit gegeben, welche den weiteren Ablauf bestimmt. Entweder werden jetzt die `ToDo's` direkt ausgeführt (Löschen) oder es wird z. B. eine Web-Seite mit einer Editor-Maske aufgerufen. Diese wiederum beinhaltet im `<target>`-Tag das zu benutzende Verarbeitungs-Servlet, welches dann je nach Aufgabe wieder zu einer Web-Seite oder der Workflow-Ansicht verzweigt.

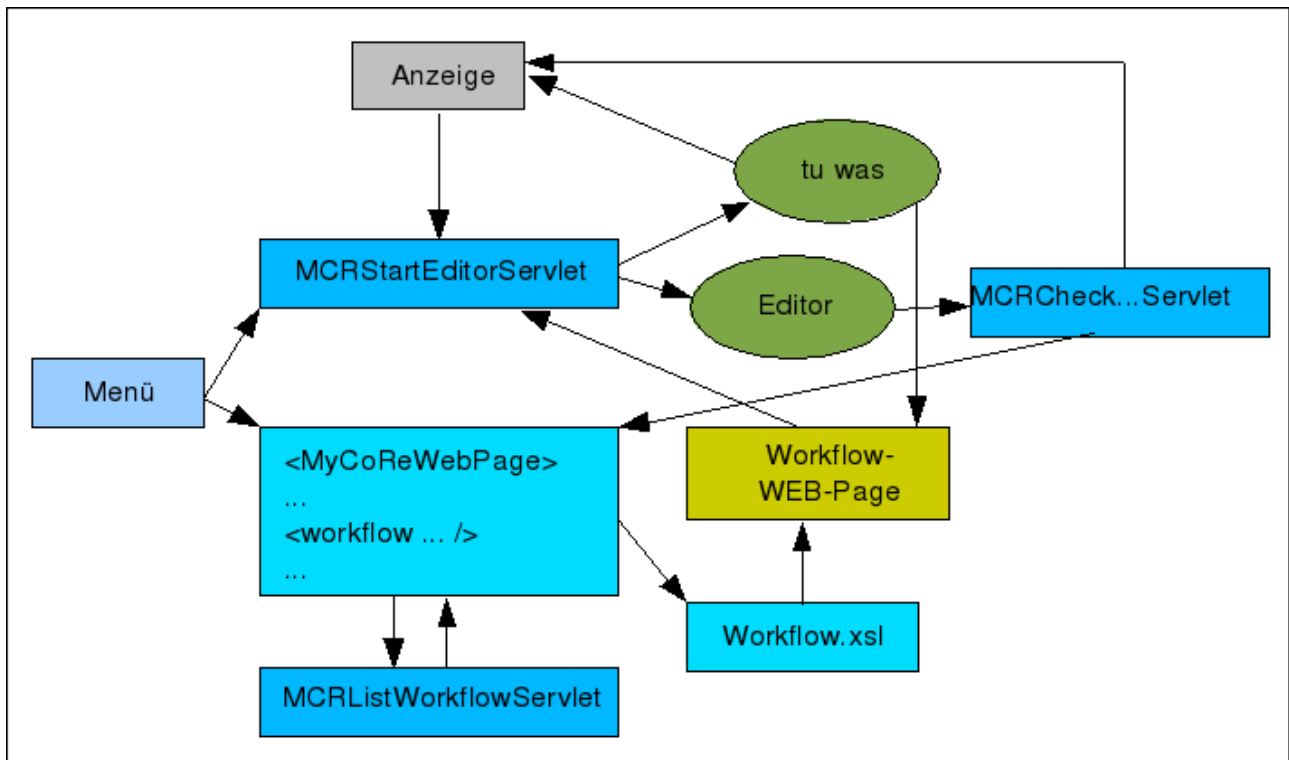


Abbildung 4.2: Ablaufschema im SimpleWorkflow

### 4.1.3 Installation

In DocPortal ist ein großer Teil der Funktionen bereits integriert. Bei der Verwendung in anderen Applikationen sind wieder die zwei Funktionskomplexe zu unterscheiden – für die einfache Workflow-Gestaltung auf der Platte müssen sie nur:

1. ... das *workflow.xml* Stylesheet in Ihre Web-Applikation kopieren. Da das Stylesheet in der Darstellung der Daten sehr anwendungsspezifisch ist, sollten Sie es an das Design Ihrer Applikation anpassen. Die im Kern mitgelieferte Version kann nur als Vorlage dienen!
2. ... die Folgende Zeile in das XSL-Stylesheet kopieren, welches die Auswertung Ihrer XML-Web-Seiten realisiert (z. B. *MyCoReWebPage.xml*).

**<xsl:include href="workflow.xml" />**

3. Weiterhin finden Sie im Kern das Stylesheet *mycoreobject-document-to-workflow.xml*. Diese ist eine Transformationsvorlage von den XML-Objekt-Metadaten in eine Workflow-interne XML Struktur. Für jeden Ihrer Metadaten-Typen muss eine solche Konverter-Datei mit Namen *mycoreobject-<type>-to-workflow.xml* in Ihrer Anwendung vorhanden sein.

4. ... die Icon-Dateien in Ihre Web-Applikation zu kopieren.
5. ... die Servlets in der Datei *web.xml* Ihrer Web-Applikation aus der mitgelieferten Datei *web.xml* zu ergänzen. Die Java-Klassen sind bereits im mycore-....jar Archiv enthalten.
6. Als letztes muss der Workflow in eine XML-Webseite integriert und diese entsprechend über Menüpunkte aufgerufen werden.

```
<MyCoReWebPage>
<section title="Bearbeiten von Objekten aus dem Workflow" xml:lang="de">
<table id="contentArea" width="100%" cellspacing="20" cellpadding="0">
<tr>
<td>
<center>
<span class="textboldnormal"><u>Bearbeiten und Verwalten von Dokumenten<br />(nur für berechtigte Editoren)</u></span>
<p></p>
<workflow type="document" step="editor" />
</center>
</td>
</tr>
</table>
</section>
</MyCoReWebPage>
```

Abbildung 4.3: Beispiel-Web-Seite für den Aufruf eines Workflow

Die Integration des SimpleWorkflow in die Präsentationsseiten erfolgt unter Einbeziehung der bereitgestellten Icons und eins dahinter liegenden Link oder Formular. Dieses kann an beliebiger Stelle in der Präsentation platziert werden.

```
<a href="{${ServletsBaseURL}MCRStartEditorServlet{${HttpSession}?tf_mcrId={${id}}&se_mcrId={${id}}&type={${type}}&step=commit&todo=seeditobj}">
  
</a>
```

Abbildung 4.4: Beispiel für den Aufruf einer Aktion aus einer Web-Seite mit Daten aus dem Server

#### 4.1.4 Konfiguration

Die Konfiguration des SimpleWorkflow beschränkt sich auf einige wenige Dinge. Anzugeben sind:

- MCRObjektID-Projektnamen für die einzelnen Metadatentypen,

- die Verzeichnisnamen des Plattenspeichers,
- die Mail-Verteilung und
- die Lokation für das nachladen von Informationen.

```
#####
# SimpleWorkflow                                #
#####
# The project ID
MCR.default_project_id=DocPortal
MCR.default_project_type=document
MCR.document_project_id=DocPortal
...
# Editor path
MCR.editor_document_directory=/home/mcradmin/docportal/workflow/document
...
# Editor Mail addresses for Messages add1@serv1,add2,@serv2,...
# Editor flags for todo and type
MCR.editor_document_wnewobj_mail=user@domain.xyz
MCR.editor_document_weditobj_mail=
MCR.editor_document_wcommit_mail=user@domain.xyz
MCR.editor_document_wdelobj_mail=
MCR.editor\_document\_seditobj\_mail=user@domain.xyz
...
MCR.editor_mail_sender=user@domain.xyz
MCR.editor_mail_application_id=MyCoReSample
# Generic mail configuration for MCRMailer
MCR.mail.server=myhost.de
MCR.mail.protocol=smtp
MCR.mail.debug=false
# Editor HostAlias for Classification
MCR.editor_baseurl=local
Abbildung 4.5: Abschnitt der Konfiguration für den SimpleWorkflow
```

Hier noch einige Hinweise: Kopieren Sie den Abschnitt in Ihre Datei *mycore.properties.private* und ergänzen Sie die Einträge mit den Daten Ihrer Anwendung. Ein Teil der vom *MCRStartEditorServlet* veranlassten Aktionen ist so implementiert, dass sie auf Wunsch eine Mail an eine oder mehrere Adressen schicken können. Wenn Sie für den Konfigurationswert, welcher durch das Paar **type\_todo** beschrieben wird, nichts angeben, so wird die Mail unterdrückt. Alle

---

Angaben in diesem Konfigurationsabschnitt sollten selbsterklärend sei.

## 5. Anmerkungen und Hinweise

### 5.1 Ergänzung der DocPortal-Beispieldaten

Mit Version 1.1 wurden die Beispieldaten für das DocPortal aus der Distribution des selbigen herausgelöst und in eine separaten CVS-Baum untergebracht. Dies hat den Vorteil, dass

- die Installation des DocPortals nicht mehr von Beispieldaten abhängig ist,
- man nach der Installation ein leeres, betriebsbereites System hat,
- Die Distribution des Samples schlanker und der Download damit schneller ist,
- mehr Beispiele in einer extra-CVS-Distribution angeboten werden können und
- die Beispiele gezielt geladen und auch wieder entfernt werden können.

Die Beispieldaten stehen auf dem CVS-Server in Essen<sup>15</sup> in einem extra CVS-Baum mit dem Namen **content** bereit. Dieser Enthält eine Sammlung einzelner Beispieldatengruppen. Nach dem checkout können die Gruppen je nach Wunsch einzeln installiert werden.<sup>16</sup> Dabei spielt das jeweils mitgelieferte build.xml-Script eine wichtige Rolle, hier sind alle Funktionen zur Arbeit mit dem Beispiel definiert.

Um neue Beispieldaten bereitzustellen gibt es zwei Wege: es wird eine Beispielgruppe mit Daten ergänzt oder es wird einen neue Beispielgruppe aufgebaut.

#### 5.1.1 Ergänzungen in einer Beispielgruppe

Folgende Arbeiten sind erforderlich:

- Erzeugen der Metadaten für das Dokument (ggf. mit Daten für den Autor und/oder die Institution).
- Erzeugen des/der Derivate.
- Integration des Ladens und Entfernens im build.xml Script in den target's **load...** und **remove...**

#### 5.1.2 Hinzufügen einer neuen Beispielgruppe

Hier sind mehr Schritte erforderlich. Dabei ist immer darauf zu achten, dass die beispielgruppe in sich vollständig ist, d. h. alle Autoren- und Institutionsdaten mitgeliefert werden. Da zu Laden der Daten das Update-Kommando verwendet wird, ist sichergestellt, dass es keine Dopplung im System gibt. verwenden Sie möglichst die schon vorhandenen Autoren und Institutionen erneut.

- Checken Sie die **content**-Distribution aus.
- Erzeugen Sie ein neues Verzeichnis unter **content**, welches die Wurzel für Ihre Beispielgruppe sein soll.
- Übernehmen Sie die Verzeichnisstruktur von einem bestehenden Beispiel und füllen Sie diese mit Ihren Daten.
- Kopieren Sie das build.xml-Script von einem bestehenden Beispiel und adaptieren Sie es. Es

---

<sup>15</sup> server.mycore.de

<sup>16</sup> siehe UserGuide

muss mindestens die tagret's **info**, **load** und **remove** beinhalten. Ggf. sind noch weitere target's zum Kopieren von Stylesheets usw. nötig, hier müssen Sie das build.xml-File ergänzen.

- Schreiben Sie ein kurzes ReadMe-File im ASCII-Format mit Installationshinweisen.
- Testen Sie das fertige Beispiel.
- Comitten Sie alles in den CVS-Server.

```
content
+-- mein_sample
+-- build.xml
+-- readme.txt
+-- document
+-- derivate-Verzeichnisse (optional)
+-- author (optional)
+-- institution (optional)
```

Abbildung 5.1: Mindeststruktur einer Beispielgruppe