JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Evaluierung von Web-Frameworks, Erstellung eines Prototyps und Konzeption der multimedialen Wissensdokumentation für die Kunstuniversität Linz

Diplomarbeit zur Erlangung des akademischen Grades einer

Diplom Ingenieurin

in der Studienrichtung Informatik

Angefertigt am Institut für Bioinformatik

Betreut von a.o. Univ.-Prof. Mag. Dr. Werner Retschitzegger

Eingereicht von Sabine Daschiel

Linz, 24. Jänner 2005

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplom- bzw. Magisterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Dezember 2004

Danksagung

Hiermit möchte ich all jenen DANKE sagen, die zum Gelingen dieser Diplomarbeit direkt und indirekt beigetragen haben.

Meinen Eltern, die mich ohne Druck arbeiten ließen und großes Vertrauen in mich gesetzt haben.

Meiner Cousine Mag.[a] Claudia Koch, die mich durch etliche Balancen und Behandlungen energetisch immer wieder ins Gleichgewicht gebracht hat.

Meinen Studienkollegen DI Bernhard Lackner und Martin Guttenbrunner, die mich durch mein Studium begleitet haben und mit denen ich wirklich gerne zusammengearbeitet habe.

Meinen Freunden Lisa Filzmoser, Sabine Neubauer und Dominik Sonnberger, die mich in der für mich schwierigen Zeit motiviert und in meinem Wesen bestärkt haben.

Ursula Steininger, die mit mir analysiert hat, wo genau die Hindernisse zu meinem Ziel liegen und mit mir Lösungsmodelle gesucht hat, wie ich diese am Besten beseitigen kann.

Mag. Johannes Koch und Mag. Manfred Lechner, durch die ich zu meinem Diplomarbeitsthema gekommen bin.

Mag. Gregor Traugott, Markus Decker und DI Herwig Wiesinger, die mich bei der technischen Umsetzung unterstützt haben.

Ganz besonders möchte ich aber meinem Professor Dr. Werner Retschitzegger danken, der mich nicht nur fachlich sondern auch mental unterstützt hat und durch dessen professionelles Feedback schließlich auch ich mit meiner Arbeit zufrieden bin.

Kurzfassung

Durch die steigende Autonomie der Universitäten, wird es für diese immer wichtiger, ihre Forschungsergebnisse öffentlich zu präsentieren. Deshalb stellen Universitäten mehr und mehr eigene Leistungen mit Hilfe einer Forschungsdokumentation über das Internet zur Verfügung. Gemäß dieser Anforderung ist es auch Ziel der Kunstuniversität Linz ihre vorwiegend multimedialen Forschungsergebnisse in geeigneter Form online zu dokumentieren und der Öffentlichkeit über das Internet zur Verfügung zu stellen. Ziel dieser Diplomarbeit ist die Konzeption und Entwicklung eines Prototyps einer multimedialen, web-basierten Wissensdokumentation für die Kunstuniversität Linz. Besonderes Augenmerk wurde dabei auf die Auswahl eines geeigneten Java Web-Frameworks gelegt, um die Wartungsfreundlichkeit und Erweiterungsfähigkeit des Prototyps zu erhöhen.

Dazu werden im ersten Teil der Arbeit fünf - in Bezug auf die unterstützten Konzepte und den Grad ihrer Verbreitung - repräsentative Frameworks (Struts, Spring, Maverick, WebWork2 und Tapestry) in die engere Wahl gezogen. In einem weiteren Schritt wird ein umfassender Kriterienkatalog zur detaillierten Evaluierung dieser Frameworks aufgestellt, der nicht nur verschiedene Aspekte des MVC-Paradigmas umfasst sondern darüber hinaus auch Kriterien wie Formular-Validierung, Daten-Bindung, Internationalisierung oder anwendungsspezifische Aspekte beinhaltet. Die Ergebnisse der Evaluierung zeigen, dass insbesondere in Bezug auf Konzepte zur Daten-Bindung und Wizard-Formular-Unterstützung aber auch hinsichtlich der Konzepte zur deklarativen Transaktions-Verwaltung, Datenbank-Anbindung und Verwaltung von Objekt-Abhängigkeiten das Framework Spring vor den anderen betrachteten Frameworks zu reihen ist.

Im zweiten Teil der Arbeit wird die Funktionalität des realisierten Prototyps sowie dessen Architektur und Implementierung auf Basis des ausgewählten Frameworks Spring vorgestellt. Der Prototyp stellt im Wesentlichen Funktionen zur Verwaltung von Benutzern und deren Autorisierung zur Verfügung sowie Funktionen zur Erfassung und Suche von Forschungsergebnissen und sieht Möglichkeiten zur Qualitätskontrolle der erfassten Daten durch einen expliziten Freigabemechanismus und den Export der Daten in andere Systeme vor.

Abstract

Because of their increasing autonomy, universities have to put more effort in informing the public about their work. That's why they have started to use a research documentation tool to present their research results via the Internet. According to this requirement the University of Art in Linz would also like to document their mainly multimedia based results in a suitable way and provide them via the Internet.

The aim of this diploma thesis is to develop both concept and prototype for the web based research documentation, which can handle multimedia objects as well as metadata. Therefore a lot of attention was turned on the choice of Java Web-Frameworks to increase maintainability and extensibility of the prototype.

In the first part of this work five representative frameworks – concerning their concepts and spreading - are selected (Struts, Spring, Maverick, WebWork2 and Tapestry). To evaluate these frameworks in detail, an extensive criteria catalogue is developed in the next step, which contains not only aspects of the MVC paradigm but also criteria like validation, data-binding, internationalism and application specific ones. The results of this evaluation show, that it is more appropriate to use Spring than the other frameworks, not only because of it's support for wizard forms and it's concept of data-binding, but also because of the provided declarative transaction management, JDBC abstraction and "Inversion of Control" container.

In the second part of this work functionality as well as architecture and implementation of the developed prototype based on the chosen framework Spring is described. The prototype provides user management and authorisatic functions as well as functions to collect and retrieve research results and gives the possibilities to control the quality of the collected data through an explicit release mechanism and to export the results in other systems.

Inhaltsverzeichnis

1. Kapitel

Einleitung

1.1 Motivation

Durch die steigende Autonomie der Universitäten, wird es für diese immer wichtiger, ihre Forschungsergebnisse öffentlich zu präsentieren. Deshalb stellen Universitäten mehr und mehr ihre Leistungen mit Hilfe einer Forschungsdokumentation über das Internet zur Verfügung. Die Johannes Keppler Universität Linz besitzt seit 1997 eine webbasierte Forschungsdokumentation namens FoDok [Wies03]. Diesem Beispiel folgend möchte auch die Kunstuniversität Linz ihre Forschungsergebnisse öffentlich über das Internet zugänglich machen. Die Inhalte der Forschung und Lehre sollen in geeigneter Form dokumentiert und der Öffentlichkeit über das Internet zur Verfügung gestellt werden, denn Transparenz der Forschung und Lehre ist ein wichtiger Wettbewerbsfaktor.

Es kann jedoch nicht die FoDok der JKU Linz verwendet werden, da diese nur für textbasierte Forschungsarbeiten konzipiert ist und somit nicht für die vorwiegend multimedialen Forschungsergebnisse der Kunstuniversität geeignet ist. Es ist aber ein längerfristiges Ziel die Forschungsergebnisse aller Linzer Universitäten durch ein gemeinsames System zur Verfügung zu stellen.

Der Name für das Forschungs- und Wissensdokumentations-Projekt der Kunstuniversität Linz ist mm:widok, der als Abkürzung für „multimediale Wissensdokumentation“ zu verstehen ist. Da die Informationen über die Leistungen

der Kunstuniversität Linz über das Internet verbreitet werden sollen, bietet sich eine webbasierte Lösung für dieses Projekt an.

Alle Web-Applikationen stehen vor den gleichen Problemstellungen und Herausforderungen einer sauberen Implementierung. Code-Wiederverwendung ist nicht zuletzt deshalb zu einem zentralen Programmier-Prinzip geworden. Da nicht bei jeder Web-Anwendung das Rad neu erfunden werden soll, haben sich in den letzten Jahren immer mehr Frameworks entwickelt, die die Implementierung von Web-Applikationen unterstützen und bewährte Lösungen für die wiederkehrenden Problemstellungen bieten [Ford04]. Obwohl der Begriff Web-Framework streng genommen nicht ganz korrekt ist, da ja nicht das Web, sondern eine Web-Anwendung gemeint ist, wird dieser sowohl in der Literatur als auch in dieser Arbeit verwendet.

Mit dem steigenden Framework-Angebot steigt jedoch auch die Schwierigkeit, sich für eines zu entscheiden. Es sind momentan ungefähr 50 Open-Source Java Web-Frameworks verfügbar [Tomp04]. Diese Frameworks sind sowohl in ihrem Umfang und als auch in der verwendeten Technologie sehr verschieden, so dass man nicht auf Anhieb eine Auswahl treffen kann.

1.2 Ziele und Aufbau dieser Arbeit

Auf Grund der in Abschnitt 1.1 erwähnten Gründe gibt es in dieser Arbeit zwei Hauptziele: Erstens die Durchführung einer Evaluierung von Web-Frameworks und zweitens die Entwicklung eines Java basierten Prototyps für die multimediale Wissensdokumentation der Kunstuniversität Linz.

Gemäß dieser Ziele ergibt sich der Aufbau dieser Arbeit wie folgt: Zunächst werden in Kapitel 2 sowohl sämtliche Arbeitsschritte erläutert, die der eigentlichen Evaluierung vorausgehen, als auch das den Frameworks zu Grunde liegende „Model 2“-Konzept erklärt. Danach beschreibt Kapitel 3 die Evaluierungs-Kriterien, nach denen bewertet wird. Kapitel 4 bis 8 enthalten die Beschreibungen der einzelnen Frameworks. In Kapitel 9 werden die entstandenen Evaluierungsergebnisse noch einmal zusammengefasst und einander gegenübergestellt. Danach gibt Kapitel 10 Aufschluss über Funktionalität und Implementierung des Prototyps. Abschließend folgt in Kapitel 11 ein Ausblick.

2. Kapitel

Grundlagen

Dieses Kapitel beginnt mit einem Abschnitt über das „Model 2“-Konzept, einem Architekturmuster für Web-Applikationen, das allen Java basierten Web-Frameworks zu Grunde liegt. Anschließend wird kurz erläutert, welche Frameworks für diese Evaluierung ausgewählt wurden und wie die Beispiel-Applikation aufgebaut ist, die zur Veranschaulichung der Verwendung der einzelnen Frameworks dient.

2.1 Das „Model 2“-Konzept

Das „Model 2“-Konzept [Sun04b] ist ein Vorschlag für die Umsetzung der Präsentationslogik, die eine strikte Trennung der Schichten einer Web-Applikation sicherstellt. Diese Trennung erfolgt im Sinne einer Drei-Schichten-Architektur, die eine Web-Anwendung in drei von einander unabhängigen Schichten (Präsentations-, Geschäfts- und Datenbankzugriffslogik) einteilt. Charakteristisch dabei ist, dass jede Schicht immer nur die direkt unter ihr liegende kennt und dass die Kommunikation lediglich über eindeutig definierte Schnittstellen erfolgt.

Um die Trennung verwirklichen zu können, stützt sich dieses Architekturmuster auf das Model-View-Controller-Prinzip und setzt dieses mit Java-Klassen um. Die daraus resultierende Kompetenzenverteilung bewirkt, dass Änderungen an der View den Controller-Code nicht beeinflussen. Das „Model 2“-Konzept sieht vor, dass die darzustellenden Daten als JavaBeans vorliegen, Servlets [Sun03b] und Helferklassen den Kontrollfluss steuern und Java Server Pages [Sun03a] die Präsentation über-

nehmen. Natürlich kann dieses Muster auch mit anderen View-Technologien angewandt werden. Abb. 2-1 zeigt das typische „Model 2“-Szenario:

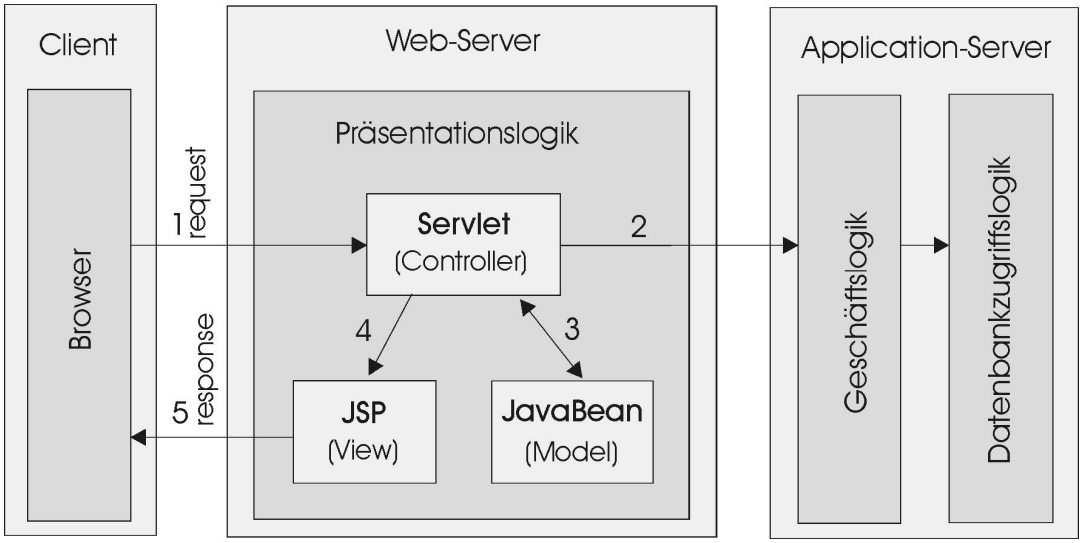

Abb. 2-1: „Model 2“ –Architektur

Im Folgenden werden die in Abb. 2-1 abgebildeten Komponenten und ihr Zusammenspiel innerhalb eines Web-basierten Szenarios näher erläutert: Der Controller behandelt die an ihn gerichteten Anfragen (1) und reagiert darauf. Die aus dem Request extrahierten Daten gibt er weiter an die Methoden der Geschäftslogik (2), behandelt die eventuell daraus resultierenden Ausnahmen, befüllt das Model mit den Ergebnissen (3) und wählt eine passende View aus (4). Der Controller ist die einzige der drei Komponenten, die Zugriff auf die Geschäftslogik hat [John03].

Die View stellt die Daten dar, die ihr im Model zur Verfügung gestellt werden. Dazu braucht sie nichts über die Implementierung der Geschäftslogik zu wissen. Sie ist allein für die Markup-Generierung und die Darstellung des Model-Inhalts verantwortlich (5). Es gehört weder zu ihrem Aufgabengebiet Datenbankabfragen durchzuführen, noch muss sie sich mit Fehlerbehandlungen auseinandersetzen. Eine View sollte auch nicht direkt mit Request-Parametern arbeiten, da das die Aufgabe des Controllers ist. Den einzigen Code den eine View enthalten darf ist Präsentations-Logik [John03].

Die Aufgabe des Models ist es, die Daten zu verwalten, die von der View dargestellt werden sollen. Da es keine weiteren Aufgaben hat, sollte das Model als einfacher Datencontainer umgesetzt werden, der eine - sowohl von der View als auch vom Controller unabhängige - Komponente darstellt. Deshalb ist das Model gewöhnlich eine JavaBean, die neben den Attributen, nur Zugriffs- und Änderungsmethoden aufweist [John03].

2.2 Auswahl der Frameworks

Aus den in [diNi04] angeführten Frameworks wurden folgende fünf ausgewählt, um in dieser Arbeit genauer auf ihre Funktionalitäten untersucht zu werden:

Struts, da es das älteste und meist verwendete Framework ist [John03], das es derzeit gibt und somit bei einem Vergleich nicht ausgeschlossen werden kann.

Spring, weil es moderne Konzepte wie aspektorientierte Programmierung aufweist und Wizard-Unterstützung bietet [John03] [Whisper].

Maverick, weil es einfach und leicht zu lernen ist und dennoch die Basis-Funktionalitäten eines Web-Frameworks bietet [John03] [Tho03a].

WebWork2, da auch dieses Framework aspektorientierte Programmierung aufweist, jedoch die Controller nicht als Singletons realisiert sind, wie das bei *Spring* der Fall ist [Ford04] [Tho03b] [Whisper].

Tapestry, um auch ein ereignisgesteuertes Framework im Vergleich zu haben, im Unterschied zu den anderen, die einen aktionsgesteuerten Ansatz verfolgen [Ford04].

In den folgenden Kapiteln werden diese fünf Frameworks beschrieben und auf Basis der in Kapitel 3 beschriebenen Evaluierungskriterien verglichen. Zum besseren Verständnis werden Code-Beispiele die beschriebenen Funktionalitäten begleiten. Diese Beispiele sind Auszüge einer bewusst einfach gewählten Beispiel-Applikation, die schon in dem Buch „Art of Java Web Development“ von Neal Ford [Ford03] zur Illustrierung dient. Diese Beispiel-Applikation wurde auch in dieser Arbeit verwendet und für alle fünf Frameworks teils aktualisiert und teils neu implementiert. Im folgenden Abschnitt wird diese Beispiel-Applikation kurz vorgestellt.

2.3 Beispiel-Applikation

Die Applikation realisiert eine Terminverwaltung, deren sehr einfache Präsentation von nur zwei Webseiten übernommen wird. Die erste Seite zeigt eine Liste aller Termine (siehe Abb. 2-2), während die zweite Seite das Hinzufügen von neuen Terminen ermöglicht (siehe Abb. 2-3).

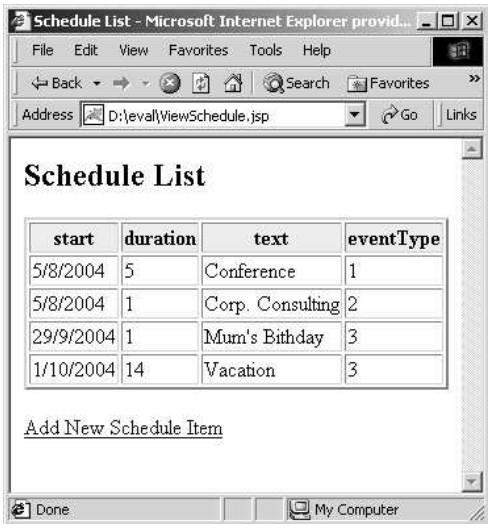

Abb. 2-2: ScheduleView.jsp

Abb. 2-2 zeigt die erste Seite der Applikation. In dieser werden unterhalb des Namens „ScheduleList“ alle in der Datenbank vorhandenen Termine angezeigt. Es wird angeführt, wann ein Termin beginnt, wie lange er dauert, wie er heißt und von welchem Typ er ist. Über den Link „Add New Schedule Item“ gelangt man zur zweiten Seite der Beispiel-Applikation *SchedlueEntryView*, die in Abb. 2-3 gezeigt wird:

Abb. 2-3: ScheduleEntryView.jsp

Dieses Formular ermöglicht das Hinzufügen eines neuen Termins. Es fordert zur Eingabe der Dauer, des Starts, des Namens und zur Auswahl der Art des Termins auf. Mit dem Button *submit* wird das Formular abgeschickt und vom zuständigen Controller bearbeitet, der auch die Felder validiert. Sind alle Formularfelder richtig eingegeben worden, das heißt, alle Felder sind ausgefüllt und *Duration* ist eine Zahl zwischen 0 und 31, wird die Kontrolle wieder der Startseite (Abb. 2-2) übergeben, in der auch der neue Termin angezeigt wird. Mit dem Button *reset*, werden alle Einträge gelöscht. Der Link *back* führt ohne Aktion wieder zur *ScheduleView*-Seite.

Das Model dieser Applikation ist das *ScheduleItem*, ein Objekt das einen Termin darstellt (siehe Abb. 2-4). Seine Attribute korrespondieren mit den Formularfeldern. Nach dem Abschicken des Formulars wird ein neues Objekt der Klasse *ScheduleItem* erzeugt und dessen Attribute werden mit den Request-Parameter-Werten befüllt. Sind bei der Validierung keine Fehler aufgetreten wird das *ScheduleItem* in die Datenbank gespeichert. Diese Aufgabe übernimmt die Klasse *ScheduleDb*.

In Abb. 2-4 sind alle für diese Beispiel-Applikation zu implementierenden Klassen in einem Klassendiagramm dargestellt:

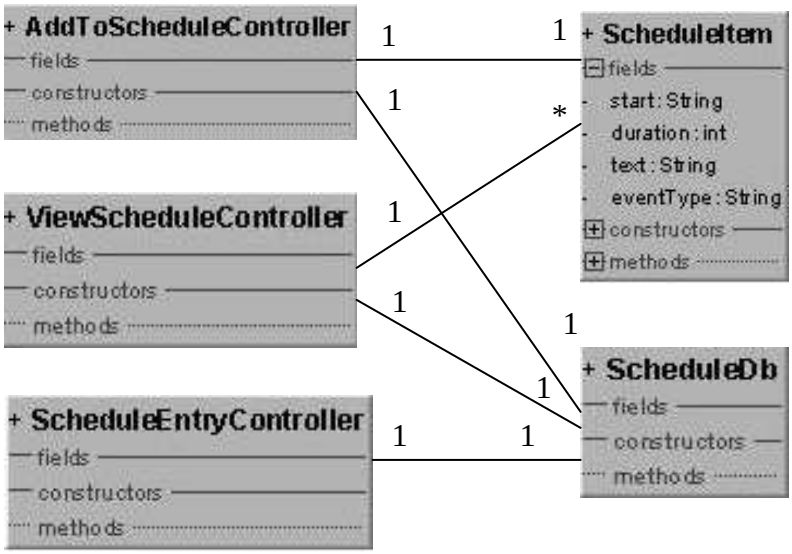

Abb. 2-4: Klassendiagramm der Beispiel-Applikation

In Abb. 2-4 sind links die drei Controller zu sehen, die die Präsentationslogik der Beispiel-Applikation darstellen. Es sind drei Controller notwendig: je einer zur Darstellung jeder Seite (*ViewScheduleController* und *ScheduleEntryController*) und einer für die Formularverarbeitung (*AddToScheduleController*).

Die beiden Klassen *ScheduleItem* und *EventType* repräsentieren die Geschäftslogik. Mit *ScheduleItem* wird ein Termin realisiert. Die verschiedenen Termintypen sind in einer Tabelle *sched_eventtypes* gespeichert.

Die Klasse *ScheduleDb* bildet die Datenbankzugriffslogik, deren Aufgabe es ist, das *ScheduleItem* in die Datenbank zu speichern. Die Schnittstelle von *ScheduleDb* wird in Abb. 2-5 dargestellt:

```
public class ScheduleDb{
   public void addRecord(ScheduleItem si) {…}
   public List getEvents() {…}
   public List getEventTypes(){…}
}
```

Abb. 2-5: ScheduleDb

Mit der Methode *addRecord()* wird ein *ScheduleItem*, das die Daten eines Termins enthält, in die Datenbank gespeichert. Die *getEvents()*-Methode ist für das Auslesen der bereits vorhandenen Termine aus der Datenbank zuständig, *getEventTypes()* liefert die verschiedenen Termintypen, die zur Auswahl stehen.

3. Kapitel

Evaluierungs-Kriterien

In diesem Abschnitt werden die Evaluierungs-Kriterien diskutiert, mit deren Hilfe aufgezeigt werden soll, welches Framework am besten als Entwicklungsbasis für die multimediale Wissensdokumentation geeignet ist. Diese sollen gleichzeitig als Strukturierungsmittel für die Beschreibung der in der Evaluierung betrachteten Web-Frameworks dienen. Dazu werden die Kriterien in entsprechende Kategorien eingeteilt.

Die in dieser Arbeit verwendeten Evaluierungs-Kriterien sind aus dem Studium verschiedener Quellen entstanden (siehe z.B. [John02], [Wafer], [McLa00], [Eagl04], [Ford04]). Es wurden jedoch nur jene Kriterien für den Vergleich herangezogen die eindeutig und objektiv überprüfbar sind und für die vorliegende Problemstellung der Realisierung einer multimedialen Wissensdokumentation besonders relevant erscheinen. Deshalb sind die von [McLa00], [Eagl04] und [Ford04] geforderten Eigenschaften wie z.B. Stabilität und Erweiterbarkeit nicht als Evaluierungs-Kriterien für diese Arbeit aufgenommen worden.

Rod Johnson schreibt in „J2EE Design and Development“ [John02], dass bei der Entwicklung von Web-Applikationen sowohl auf eine saubere als auch auf eine dünne Präsentationsschicht zu achten ist. Diese Forderungen implizieren Anforderungen an Controller (Abschnitt 3.1), Model (Abschnitt 3.2) und View (Abschnitt 3.3). Auch Neal Ford weist in seinem Buch „Art of Java Web Development“ [Ford04] darauf hin, dass die Umsetzungen von Design Prinzipien, wie eine saubere Schichtentrennung und

eine klare Rollenverteilung innerhalb der Präsentationsschicht ein Framework auszeichnen.

Formular-Validierung und Daten-Bindung sind zwei Bereiche, in denen ein Web-Framework die Implementierung einer Web-Applikation erheblich erleichtern kann. Deshalb wird auf deren Unterstützung in [John02] und im Wafer-Project [Wafer04] großen Wert gelegt. Und auch in dieser Arbeit stellen sie zwei wesentliche Kategorien von Evaluierungs-Kriterien dar (Abschnitt 3.4 und 3.5). Aus dem Wafer-Projekt, das auf den Vergleich verschiedene Frameworks abzielt, wurden außerdem noch folgende Kriterien übernommen: Unterstützung verschiedener View-Technologien (Abschnitt 3.2), Internationalisierung (Abschnitt 3.6) und Fehlerbehandlung (Abschnitt 3.5).

Natürlich hat jede Web-Applikation auch ihre individuellen Eigenheiten, die bei der Evaluierung eines Frameworks berücksichtigt werden müssen [Ford04]. Für die multimediale Wissensdokumentation der Kunstuniversität Linz ergeben sich dadurch folgende anwendungsspezifische Anforderungen, die sich in der Kriterienkategorie in Abschnitt 3.7 wieder finden: Wizard-Formular-Unterstützung, Datei-Upload-Unterstützung, deklarative Transaktionsverwaltung und JDBC-Abstraktion. Ein weiteres Kriterium dieser Kategorie ist die Verwaltung der Abhängigkeiten von Applikationsobjekten durch dynamisches Laden und Verknüpfen von JavaBeans, da dies eine lose Kopplung ermöglicht und sich so positiv auf Flexibilität, Skalierung, Erweiterung und Wartbarkeit auswirkt [John03].

Ein Kriterium, das sowohl in [Wafer] als auch in [Ford04] aufscheint, ist die Dokumentations-Qualität. Dieses wurde mit dem Abschnitt 3.8 in den Kriterien-Katalog aufgenommen.

Im Folgenden werden diese acht Kriterienkategorien näher beschrieben. Die Kriterien sind von K 3.1-1 bis K 3.8-3 durchnummeriert. So kann im Zuge der Evaluierung auf die einzelnen Kriterien Bezug genommen werden. Der erste Teil der Nummerierung, der sich zwischen dem Buchstaben K (Kriterium) und dem Bindestrich befindet, enthält die Abschnittsnummerierung der zugehörigen Kriterienkategorie.

3.1 Controller

Das Framework soll einen Rahmen bieten, mit dem es einfach ist, das „Model 2“-Konzept und somit das MVC-Entwurfsmuster zu realisieren. Der Controller sollte in diesem Rahmen folgende Eigenschaften aufweisen:

- Der Controller ist eine JavaBean, um eine einfache und konsistente Konfiguration zu ermöglichen. (K 3.1-1)
- Der Kontrollfluss einer Formularseite unterscheidet sich von dem einer Seite die nur Informations-Darstellung zur Aufgabe hat. Deshalb stellt das Framework verschiedene Superklassen für die Umsetzung der zwei unterschiedlichen Kontrollflüsse zur Verfügung. (K 3.1-2)
- Der Controller ist ein Objekt der Präsentationslogik. Datenbankzugriffe gehören nicht zu seinem Zuständigkeitsbereich. Um das Model mit Daten aus einer Datenbank zu befüllen greift er auf Dienste der Geschäftslogik zu. Der Controller enthält auch keinerlei Informationen, die sich auf Datenbanken beziehen. (K 3.1-3)

3.2 Model

Da das Model sowohl für die Geschäftslogik als auch für die Präsentationslogik wichtig ist, muss das Model für beide Schichten einsetzbar sein.

- Das Model ist Servlet- und Framework-unabhängig, damit es an die Geschäftslogik übergeben werden kann. (K 3.2-1)
- Das Model ist ein Datencontainer der mehrere JavaBeans der Geschäftslogik enthalten kann. (K 3.2-2)
- Das Model ist durch eine eigene Komponente realisiert. (K 3.2-3)

3.3 View

In jeder Web-Anwendung ist es notwendig, dass der Controller die entsprechende View auswählt, die im Browser angezeigt werden soll. Er muss ihr die notwendigen Model-Daten zur Verfügung stellen und sie veranlassen, sich darzustellen. Diese Funktionalität sollte in der Implementierung einer Web-Applikation nicht mehr notwendig sein, sofern ein Framework verwendet wird [John03]. Oft muss sich eine Anwendung in die bestehende Applikations-Landschaft eines Unternehmens eingliedern und ist deswegen an eine bestimmte View-Technologie gebunden. Ein Framework sollte deshalb in diesem Punkt flexibel sein.

- Es werden verschiedene View-Technologien unterstützt. (K 3.3-1)
- Es ist möglich, die View und auch die View-Technologie auszutauschen ohne den Controller zu ändern. (K 3.3-2)

3.4 Formular-Validierung

Formular-Validierung ist ein sehr wichtiger Teil einer Web-Applikation, da sie oft ausschlaggebend für die Akzeptanz der BenutzerInnen ist, sich aber ohne Framework-Unterstützung als sehr arbeitsaufwändig erweist. Unter Formular-Validierung wird die Überprüfung von Benutzereingaben auf ihre syntaktische und semantische Gültigkeit verstanden. Es ist deshalb so wichtig, die eingegebenen Daten zu validieren, da alle Werte, die über HTTP geschickt werden vom Typ *String* sind. Oft werden aber Datenwerte anderer Typen benötigt. Frameworks sollten hilfreiche Unterstützung für eine saubere, wieder verwendbare und erweiterbare Implementierung der Validierung geben. Es ist von Vorteil, wenn sich die Validierungsregeln außerhalb des Codes einstellen lassen [John03].

- Validatoren sind von der Servlet-API unabhängige JavaBeans, damit sie auch in der Geschäftslogik wieder verwendet werden können. (K 3.4-1)
- Das Framework stellt ein Interface zur Verfügung, das implementiert werden kann, damit eigene Validatoren erzeugt werden können. (K 3.4-2)
- Das Framework bietet deklarative Validierung. (K 3.4-3)
- Die Anzeige entsprechender Fehlermeldungen wird unterstützt. (K 3.4-4)

3.5 Daten-Bindung

Unter Daten-Bindung wird das automatische Befüllen von Bean-Properties mit den entsprechenden Request-Parameter-Werten verstanden. Dabei wird die JavaBean mit einem Formular assoziiert, oder anders ausgedrückt wird sie an das Formular „gebunden“. Die zu dem Formular gehörende JavaBean wird CommandBean genannt. Idealerweise ist die CommandBean Servlet-API und Framework unabhängig, damit sie an die Geschäftslogik weitergereicht und verarbeitet werden kann [John03].

Daten-Bindung kann mit Hilfe eines Frameworks automatisiert werden. Dazu stellt die CommandBean Properties zur Verfügung, deren Namen den Request-Parameter gleichen. Das Framework verwendet Reflection um die JavaBean mit den Request-Parametern zu befüllen. Einige Frameworks verwenden das Jarcata Commons *BeanUtil*-Package, das auch von der in Abb. 3-1 dargestellten JSP-Direktive verwendet wird.

```
<jsp:setProperty name=“bean“ property“*“ />
```

Abb. 3-1: JSP-Direktive zur Daten-Bindung

Dieser Mechanismus, um Bean-Properties mit Request-Parametern zu befüllen ist kein ausgereifter Ansatz, da es keine Möglichkeit gibt Typunverträglichkeiten zu überprüfen. Zum Beispiel wird ein Property gar nicht gesetzt, wenn der entsprechenden Parameter nicht den erwarteten Typ aufweist. Wodurch nicht mehr unterschieden werden kann, ob ein Wert mit falschem Typ oder kein Wert übergeben wurde, ohne den Request-Parameter direkt zu prüfen. Ein weiterer daraus resultierender Nachteil ist, dass dem/der BenutzerIn die falsch eingegebenen Werte nicht als Feedback angezeigt werden können. Außerdem gibt dieser Ansatz weder die Möglichkeit zwischen Pflichtfeldern und optionalen Feldern zu unterscheiden, noch auf eventuell auftretende Exceptions der CommandBeans einzugehen [John03].

- Die CommandBean ist Servlet-API- und Framework-unabhängig, damit sie ohne weiteres an die Geschäftslogik weitergegeben und dort verwendet werden kann. (K 3.5-1)
- Das Framework bietet Unterstützung bei Typüberprüfungen und kann Typkonvertierungen automatisch durchführen. (K 3.5-2)
- Es gibt eine angemessene Unterstützung für die Ausnahme-Behandlung von Fehlern, die durch die Daten-Bindung auftreten können. (K 3.5-3)
- Das Framework bietet Unterstützung für die Anzeige der zuvor eingegebenen Formulardaten, wenn das Formular auf Grund falscher oder fehlender Eingabe dem/der BenutzerIn zurückgeschickt wird. (K 3.5-4)

3.6 Internationalisierung

Internationalisierung bedeutet, dass die Seiten einer Web-Anwendung gemäß der Sprache und Kultur des Benutzers/ der Benutzerin angezeigt werden. Jeder Browser hat bestimmte Länder- und Spracheinstellungen, die er mittels der „locale“-Information im HTTP-Header für die Applikation zugänglich macht.

- Das Framework liest die Länderinformation des Browser aus dem Request aus und stellt sie der Applikation zur Verfügung. (K 3.6-1)
- Die Beschriftungen der Eingabefelder können automatisch, dem Land entsprechend, angezeigt werden. (K 3.6-2)
- Die Fehlermeldungen sind vom Java-Code getrennt, können extern konfiguriert und somit automatisch sprachengerecht angezeigt werden. (K 3.6-3)

3.7 Applikationsspezifisches

- Da die direkte Verwendung von JDBC sehr fehleranfällig ist, wird vom Framework eine API für die Datenbankanbindung zur Verfügung gestellt, mit deren Hilfe auf einer höheren Abstraktionsebene als JDBC gearbeitet werden kann. (K 3.7-1)
- Es wird der Umgang mit Transaktionen erleichtert, in dem eine Abstraktionsebene zwischen der Programmierung der Web-Applikation und den Transaktions-APIs eingeführt wird. Optimal ist es, wenn deklarative Transaktions-Verwaltung unterstützt wird. (K 3.7-2)
- Es wird die Verwaltung von Abhängigkeiten der Applikationsobjekte vom Framework übernommen, damit die Applikation flexibler und einfacher erweiterbar wird. (K 3.7-3)
- In der mm:widok kommen vor allem Wizard-Formulare zum Einsatz, da eine große Menge an Daten notwendig ist, um Forschungsarbeiten und dazugehörige Multi-Media-Objekte zu beschreiben. Deshalb bietet das Framework Unterstützung bei der Erstellung von Wizard-Formularen und erledigt Aufgaben wie die Steuerung des Kontrollflusses. (K 3.7-4)
- Diese Wizard-Formulare können auch zum Datei-Upload verwendet werden. Das Framework bietet Unterstützung beim Datei-Upload. (K 3.7-5)

3.8 Dokumentation

Die Qualität der Dokumentation eines Frameworks kann für dessen Verwendung ausschlaggebend sein.

- Die Referenz-Dokumentation ist aktuell und vollständig. (K 3.8-1)
- Die JavaDocs beinhalten sowohl auf Klassen als auch Methoden-Ebene aussagekräftige Kommentare. (K 3.8-2)
- Es stehen Beispiel-Anwendungen und Tutorials zur Verfügung. (K 3.8-3)

In den folgenden fünf Kapiteln werden die ausgewählten Web-Frameworks anhand der oben diskutierten Kriterien beschrieben und evaluiert. Da der vorliegende Kriterien-Katalog auch als Strukturierungsmittel der Beschreibung dienen soll, folgt die Abschnittsnummerierung der Kapitel über die einzelnen Web-Frameworks dem in diesem Kapitel vorgegebenen Schema. In den Abschnitten, die sich mit den anwendungsspezifischen Kriterien auseinandersetzen, werden nur die Funktionalitäten beschrieben, die vom Framework unterstützt werden.

4. Kapitel

Struts

Struts ist ein Open-Source Framework, das momentan in der Version 1.2 verfügbar ist und als ein Teil des Jarcata Projekts zur Apache Gruppe gehört [Stru04]. Es unterstützt die Entwicklung einer „Model 2“ (vergleiche 2.1) konformen Web-Applikation durch einen aktionsgesteuerten Kontrollfluss, der in Abb. 4-1 graphisch dargestellt ist:

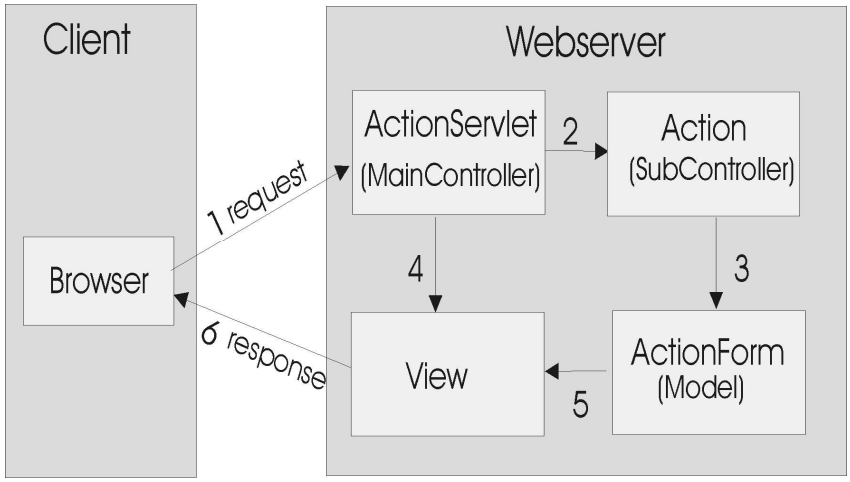

Abb. 4-1: Kontrollfluss von Struts

Der Kern von Struts ist die generische Framework-Klasse *ActionServlet*, die durch Konfigurationsdateien parametrisiert wird und den Main-Controller darstellt. Das *ActionServlet* behandelt die eingehenden Requests (1) und leitet sie an die entsprechende *Action* weiter (2), die dann die eigentlichen Controller-Aufgaben übernimmt. Eine *Action* ist somit ein Sub-Controller. Sie befüllt ein *ActionForm*-Objekt (3), das die Model-Daten für die Anzeige zur Verfügung stellt und gibt die

Kontrolle wieder an das *ActionServlet* zurück. Von diesem wird dann die richtige View ausgewählt (4). Mit Hilfe der *ActionForm*-Daten wird die View aufgebaut (5) und schließlich im Browser angezeigt (6).

4.1 Controller

Wie auch in allen anderen Frameworks, die in dieser Arbeit beschrieben werden, lagert der Main-Controller, die nicht generischen Controller-Aufgaben an Hilfs-Klassen, so genannte Sub-Controller aus. Deshalb wird die Rolle des Controllers in zwei Abschnitten – Main-Controller und Sub-Controller – beschrieben.

4.1.1 Main-Controller (ActionServlet)

In Struts stellt die generische Klasse *ActionServlet* den Main-Controller des Frameworks dar. So wie jedes Servlet, wird auch das *ActionServlet* im Deployment-Descriptor der Web-Applikation (*web.xml*) deklariert. Abb. 4-2 zeigt einen Auszug dieser Deklaration:

```
<servlet>
  <servlet-name>struts</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>application</param-name>
    <param-value>eval.struts.schedule</param-value>
  </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>struts</servlet-name>
    <url-pattern>struts/*.do</url-pattern>
</servlet-mapping>
```

Abb. 4-2: Auszug aus web.xml

Wie in Abb. 4-2 ersichtlich ist, wird im Abschnitt <*servlet*> ein Name für das Servlet vergeben und die implementierende Klasse angegeben, die schon vom Framework zur Verfügung gestellt wird. Das Servlet kann durch Init-Parameter konfiguriert werden, die mit einem „Name-Value-Paar“ spezifiziert werden. Mittels des Abschnitts <*servlet-mapping*> werden jene Requests dem *ActionServlet* zugeordnet, die von ihm

behandelt werden sollen. Es wird der zuvor spezifizierte Name des Servlets und ein URL-Muster angegeben, das beschreibt, welche URLs das Servlet erreichen können. Es werden dem Servlet meist alle Anfragen auf Dateien mit einer bestimmten Endung (z.B. do) und/oder innerhalb eines bestimmten Pfads (z.B. /struts/) zugeordnet.

Das Controller-Servlet wird schon beim Start der Web-Applikation geladen. Die Init-Parameter geben an, wo die Konfigurations-Dateien zu finden sind. Der Parameter *application* beinhaltet den Pfad zu einer Property-Datei (s*chedule.properties*), die für Internationalisierungs-Zwecke benötigt wird (siehe Kapitel 5.6). Weitere Konfigurations-Informationen für Struts sind in der *struts-config.xml* zu finden. Abb. 4-3 zeigt die *struts-config.xml*-Datei der Beispiel-Applikation:

```
<data-sources>
  <data-source
    type="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
    <set-property property="url"
                  value="jdbc:mysql://localhost/sched"/>
    <set-property property="user" value="root"  />
    <set-property property="password" value="marathon"/>
    <set-property property="driverClass"
                  value="com.mysql.jdbc.Driver"/>
  </data-source>
</data-sources>
<form-beans>
  <form-bean name="scheduleItem"
             type="eval.struts.ScheduleItem"
             dynamic="no"/>
</form-beans>
<action-mappings>
  <action   type="eval.struts.ViewScheduleAction"
            path="/viewSchedule">
    <forward name="success" path="/ScheduleView.jsp" />
  </action>
  <action  type="eval.struts.ScheduleEntryAction"
            path="/scheduleEntry">
    <forward name="success"path="/ScheduleEntryView.jsp"/>
  </action>
  <action   name="scheduleItem"
            type="eval.struts.AddToScheduleAction"
            validate="true" input="/ScheduleEntryView.jsp"
            scope="session" path="/addToSchedule">
    <forward name="success" path="/viewSchedule.do" />
    <forward name="error" path="/ScheduleEntryView.jsp" />
  </action>
</action-mappings>
```

Abb. 4-3: Auszug aus struts-config.xml

Im Abschnitt <*data-sources*> werden die Charakteristika der Datenbankverbindung angegeben. Das hat den Vorteil, dass diese Informationen außerhalb des Codes konfigurierbar sind. Aber im Sinne einer sauberen Schichtenarchitektur sollte die Präsentationslogik nichts von einer Datenquelle wissen, womit das Kriterium K 3.1-3 nicht erfüllt ist.

Die *FormBeans* werden im gleichnamigen Abschnitt <*form-beans*> definiert, das sind Objekte, die die Modelle darstellen (siehe Kapitel 5.2). Für ihre Spezifikation sind ein Name und ein Type notwendig. Weiters gibt es einen Abschnitt <*action-mappings*>, indem jedem Sub-Controller, die in Struts *Action* heißen (siehe Kapitel 5.1.2), ein Pfad zugeordnet wird. Somit weiß das *ActionServlet* welchen Request es an welchen Sub-Controller weiterleiten soll.

Im Beispiel der Schedule-Applikation gibt es eine FormBean – *ScheduleItem* – und drei Actions. *ViewScheduleAction* zeigt den Zeitplan an, *ScheduleEntryAction* zeigt das Formular zur Erfassung eines neuen Termins an und *AddToScheduleAction* fügt den neu erfassten Termin zu den bisherigen Terminen hinzu. Es ist zu bemerken, dass für eine Formularseite zwei Controller benötigt werden, da vom Framework der Formular-Kontrollfluss nicht gesondert behandelt wird. Kriterium K 3.1-2 ist damit nicht erfüllt.

Als Property der Action wird angegeben, welche View bei Erfolg und welche im Fehlerfall angezeigt werden soll (*forward name= "success"* und *forward name= "error"*). So kann das *ActionServlet* je nach Situation die richtige View anzeigen. Diese View-Konfiguration außerhalb des Codes bewirkt, dass die View fast ohne Aufwand ausgetauscht werden kann und dadurch Struts dem Kriterium K 3.3-2 gerecht wird.

4.1.2 Sub-Controller (Action)

In Struts heißen die verwendeten Sub-Controller *Actions*, diese sind wieder verwendbare JavaBeans. Deshalb wird nicht bei jedem Request ein neuer Sub-Controller angelegt, sondern jeder Controller wird nur einmal instanziert. Objekte, die gemäß diesem Ansatz verwendet werden, werden Singletons genannt. Struts-*Actions* sind somit Singletons.

Sub-Controller kapseln einen großen Teil des vom Framework zur Verfügung gestellten Verhaltens. Jeder Sub-Controller wird von der Klasse *Action* abgeleitet und überschreibt die *execute()*-Methode.

Die *execute()*-Methode bekommt als Parameter ein *ActionForm*-Objekt (siehe Kapitel 5.2), das JavaBean-Properties enthält, die mittels Reflection aus dem Request gesetzt wurden. Die *Action* reicht in dieser Methode die aus dem Request resultierenden Daten an die Geschäftslogik weiter und wartet auf die Ergebnisse. Mit Hilfe der Ergebnisse bringt sie das Model auf den neuesten Stand und definiert die View, die als nächstes angezeigt werden soll.

Abb. 4-4 zeigt die Implementierung der *execute()*-Methode eines Sub-Controllers in Struts anhand der Beispiel-Applikation, die in dieser Arbeit als Referenz-Implementierung dient (vergleiche 2.3):

```
public class AddToScheduleAction extends Action {

   public ActionForward execute(ActionMapping mapping,
       ActionForm actForm, HttpServletRequest request,
       HttpServletResponse response) throws Exception {

     ScheduleItem si = (ScheduleItem) actForm;
     ScheduleDb sb = new ScheduleDb();
     sb.setDataSource(getDataSource(request));

     try {
       sb.addRecord(si);
     } catch (ScheduleAddException sax) {
       sax.printStackTrace();
     }
     return mapping.findForward("success");
   }
}
```

Abb. 4-4: Auszug aus AddToScheduleAction.java

Die *AddToScheduleAction* erledigt in der *execute()*-Methode folgende Aufgaben: Die *FormBean* wird einem *ScheduleItem* zugewiesen, das an die Geschäfts- bzw. in diesem einfachen Fall direkt an die Datenbankzugriffslogik weitergegeben wird. Sie instanziert das Datenbankzugriffs-Objekt und ruft die Methode *addRecord()* darauf aus, die den neuen Eintrag in die Datenbank speichert. Die Methode *findForward()* des *ActionMapping*-Parameters liefert die darzustellende Seite, die dann den Rückgabewert darstellt.

In der Implementierung fallen drei Dinge auf, die nicht zu einer sauberen Schichtentrennung beitragen und die Webkomponente fest an die Geschäftslogik koppeln. Erstens übernimmt die *Action* das Setzen der Datenquelle für *ScheduleDb*, das den Datenbankzugriff kapselt, ein Sub-Controller ist aber eine Komponente der

Präsentationslogik und sollte, wenn möglich, keine Datenbank bezogenen Informationen enthalten (K 3.1-3). Zweitens wird die Abhängigkeit der Klassen *AddToScheduleAction* und *ScheduleDb* im Sub-Controller verankert, da *ScheduleDb* dort instanziert wird (K 3.7-3). Drittens ist die Klasse *ScheduleItem* vom Framework abhängig, da sie von der Klasse *ActionForm* abgeleitet wird. Deshalb wird durch die Übergabe von *ScheduleItem* an die Geschäfts- bzw. Datenbankzugriffslogik die Schichtentrennung nicht eingehalten (K 3.2-1). Dazu müsste *ScheduleItem* in ein Framework-unabhängiges Objekt kopiert werden.

4.2 Model (ActionForm)

In Struts werden nicht direkt die JavaBeans der Geschäftslogik als Model verwendet, sondern es wird eine Zwischenschicht eingeführt. Struts begründet das damit, dass die Felder eines Formulars nicht immer mit den Attributen eines Model-Objektes übereinstimmen, da ein Formular auch aus Feldern bestehen kann, die aus mehreren JavaBeans kommen, oder zusätzliche Daten enthalten kann.

Die Objekte dieser Zwischenschicht werden *FormBeans* genannt, diese werden von der Struts Klasse *ActionForm* abgeleitet. Das bedeutet, dass sie vom Framework und somit von der Servlet-API abhängig sind und nicht direkt an die Geschäftslogik weitergegeben werden sollten. Sie funktionieren also als Hüllen, die die eigentlichen Model-Objekte umgeben. Damit ist das Kriterium K 3.1-1 nicht erfüllt, da dieses Kriterium die Framework- und Webunabhängigkeit fordert.

Der positive Aspekt dieser Zwischenschicht ist, dass *FormBeans* die Eigenschaften von mehreren Model-Objekten, oder aber auch nur einen Teil eines Model-Objekts beinhalten können. Somit wird Struts dem Kriterium K 3.1-2 gerecht. Das Kriterium K 3.1-3 ist ebenfalls erfüllt, da eine *FormBean* eine eigenständige Komponente ist und keine Aufgaben des Controllers oder der View übernimmt. Abb. 4-5 zeigt die *FormBean* der Beispiel-Applikation:

```
public class ScheduleItem extends ValidatorForm{

  private String start;
  private int duration;
  private String text;
  private String eventType;
  …
```

```
 …
 public ScheduleItem(String start,int duration,
                          String text,String eventType) {
    this.start = start;
    this.duration = duration;
    this.text = text;
    this.eventType = eventType;
  }
  public void setStart(String newStart) {
    start = newStart;
  }
  public String getStart() {return start;}
  [...]
}
```

Abb. 4-5: Auszug aus ScheduleItem.java

ScheduleItem erweitert *ValidatorForm*, das von *ActionForm* erbt, damit seine Felder mit Hilfe einer XML-Datei validiert werden können (vergleiche Kapitel 5.4). Es enthält weder Datenbankzugriffs- noch Geschäftslogik. *ScheduleItem* ist ein reines Model-Objekt, das seine Felder mittels Zugriffs- und Änderungs-Methoden zur Verfügung stellt.

4.3 View

In Struts kann nicht nur JSP [Sun03a] als View-Technologie verwendet werden, sondern auch z.B. XSLT [W3C99] oder Velocity Templates [Velo04]. Kriterium K 3.3-1 wird deshalb erfüllt.

Die meiste Unterstützung bietet Struts jedoch für JSP, denn es stellt eine große Anzahl von Java-Custom-Tags zur Verfügung, die dazu dienen, Java-Code aus den JSP Seiten in „Helfer-Klassen“ auszulagern, um die JSP-Seite lesbarer und somit besser wartbar zu machen. Diese Custom-Tags werden in verschiedenen Tag-Libraries zur Verfügung gestellt.

Die *struts-bean.tld* beinhaltet Tags, die zur Beschreibung und Ausgabe-Generierung von Beans verwendet werden können. Zum Beispiel kann mit dem <*bean:message*>-Tag eine zu einer Bean gehörende Bezeichnung, die in einer Property-Datei unter einem bestimmten Schlüssel angegeben ist, ausgegeben werden.

Abb. 4-6 zeigt die Verwendung dieses Custom-Tags am Beispiel der Formularseite *ScheduleEntryView.jsp* der Schedule-Applikation:

```
<tr>
    <th align="right">
      <bean:message key="prompt.duration"/>
    </th>
    <td align="left">
      <html:text property="duration" size="16"/>
    </td>
</tr>
```

Abb. 4-6: Auszug aus ScheduleEntryView.jsp

In der Property-Datei der Schedule-Applikation gibt es einen Eintrag mit dem Schlüssel *prompt.duration*, das die Bezeichnung des Attributs *duration* angibt (siehe Kapitel 5.7). Mit Hilfe des <*bean:message*>-Tags wird die Bezeichnung aus der Datei ausgelesen und in die JSP-Seite eingefügt.

„HTML"-Tags sind in der *struts-html.tld* zu finden. Hier werden viele der Standard-HTML-Tags kopiert und mit Struts-Funktionalität aufgerüstet. Zum Beispiel gibt es das <*html:select*>-Tag für ein Listen-Element, welches das Standard-HTML-Tag <*select*> ersetzt. Abb. 4-7 zeigt die Verwendung dieses Custom-Tags am Beispiel der Formularseite *ScheduleEntryView.jsp* der Schedule-Applikation:

```
<html:select property="eventTypeKey">
  <html:options collection="eventTypes" property="value"
                labelProperty="label"/>
</html:select>
```

Abb. 4-7: Auszug aus ScheduleEntryView.jsp

Der Vorteil, den das in Abb. 4-7 gezeigte Custom-Tag mit sich bringt ist, dass durch die Angabe des *property*-Attributs das Listenelement an das Java-Objekt-Feld gebunden ist, das die FormBean darstellt. Eine weitere Erleichterung stellt das *collection*-Property dar, weil so eine ganze Liste als Auswahlfelder angegeben werden kann und keine Schleife mehr dazu notwendig ist, wie es bei dem Standard-HTML-Tags der Fall ist.

Um Scriptlets zu vermeiden, die Bedingungen oder Schleifen realisieren und so den Code schwieriger lesbar und schlecht wartbar machen, bietet Struts die *struts-logic.tld* an. Durch die Einführung der JSTL [sun04a], der Java Standard Tag Library, ist jedoch diese Bibliothek unnötig geworden.

Dadurch, dass Struts auf seine eigenen Custom-Tags aufbaut, können bei einer Trennung von Designer- und Entwickler-Rolle Probleme entstehen, da die Entwicklungsumgebungen Standard-Tags produzieren. Manche Programme, wie zum Beispiel „Dreamweaver“ bieten aber schon Erweiterungen an, mit denen Struts-Tags an Stelle der Standard-Tags automatisch generiert werden können.

4.4 Formular-Validierung

Einen interessanten Ansatz stellt die Möglichkeit dar, die Formular-Validierung deklarativ zu halten, da erstens kein Code zur Verwirklichung der Validierung notwendig ist und zweitens bei Änderungen der Validierungsregeln der Applikations-Code nicht von diesen Änderungen betroffen ist, sondern lediglich die entsprechende XML-Datei.

Struts bietet die Möglichkeit der deklarativen Formular-Validierung, die Anforderung des Kriteriums K 3.4-3 wird dadurch erfüllt. Die Validierungsregeln werden dabei in eine XML-Datei geschrieben. Dazu erweitert das Model *ValidationForm* (vergleiche Kapitel 5.2 Abb. 4-5) und in der Konfigurationsdatei von Struts wird das *ValidatorPlugIn* definiert. Abb. 4-8 zeigt diese „Plug-In“-Deklaration:

```
<plug-in
  className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
                value="/WEB-INF/validator-rules.xml,
                       /WEB-INF/validation.xml"/>
</plug-in>
```

Abb. 4-8: Auszug aus struts-config.xml

Wie Abb. 4-8 zeigt, werden in der *struts-config.xml* zwei Dokumente angegeben: *validation.xml*, die XML-Datei in der die Applikations-spezifischen Validierungsregeln vermerkt sind und *validator-rules.xml*, die generische Datei aller Struts-Applikationen.

In *validation.xml* kann angegeben werden, ob Felder Pflichtfelder sind, welche Mindest- oder Höchstlänge sie haben dürfen, von welchem Typ sie sein sollen (byte, short, integer, long, float, double, date ist möglich), in welchem Wertebereich sie liegen sollen, und ob sie speziellen regulären Ausdrücken gerecht werden sollen, wie zum Beispiel Email-Adressen, URLs oder Kreditkartennummern.

All diese Funktionen bietet das Framework standardmäßig an und es muss zu deren Verwendung nichts mehr implementiert werden. Es gibt aber nicht für jede Funktionalität einen eigenen Validator, der in der Geschäftslogik verwendet werden könnte, sondern alle Überprüfungen werden von der Klasse *FieldChecks* übernommen, das Kriterium K 3.4-1 wird nicht erfüllt. Abb. 4-9 zeigt einen Auszug aus der *validation.xml*-Datei der Beispiel-Applikation:

```
<form-validation>
  <formset>
    <FORM name="scheduleItem">
      <field property="duration"
             depends="required,integer,intRange">
        <arg0 key="prompt.duration"/>
        <arg1 name="intRange" key="${var:min}"
              resource="false"/>
        <arg2 name="intRange" key="${var:max}"
              resource="false"/>
        <VAR>
              <var-name>min</var-name>
              <var-value>0</var-value>
        </VAR>
        <VAR>
              <var-name>max</var-name>
              <var-value>31</var-value>
        </VAR>
      </field>
      [...]
    </FORM>
  </formset>
</form-validation>
```

Abb. 4-9: Auszug aus validation.xml

Abb. 4-9 zeigt, wie die Konfiguration in der *validation.xml*-Datei aussehen muss, damit das Attribut *duration* der FormBean *ScheduleItem* auf folgende drei Regeln überprüft wird: *duration* ist ein Pflichtfeld, es ist vom Typ Integer und es muss im Bereich von 0 bis 31 liegen. Zuerst wird im <*form*>-Element mit *name* das Java-Objekt referenziert, das die FormBean darstellt. Danach wird mittels des <*field*>-Elements angegeben um welches Attribut der FormBean es sich handelt (mit dem Attribut property) und nach welchen Regeln es validiert werden soll (mit dem Attribut depends). Innerhalb dieses Elements können dann die konkreten Werte, die für die Überprüfung notwendig sind, angegeben werden.

Struts übernimmt die Typüberprüfungen und die Fehleranzeige, was die Erfüllung des Kriteriums K 3.4-4 impliziert. Fehlermeldungen können zu Internationalisierungs-

Zwecken in einer Property-Datei angegeben werden (siehe Kapitel 5.6). Für die von Struts standardmäßig zur Verfügung gestellten Validatoren gibt es Standard-Fehlermeldungen [Hust02]. Abb. 4-10 zeigt in einem Auszug einer Property-Datei einige dieser Fehlermeldungen:

```
errors.required={0} is required.
errors.minlength={0} can not be less than {1} characters.
errors.maxlength={0} can not be greater than {1}
characters.
errors.integer={0} must be an integer.
errors.range={0} is not in the range {1} through {2}.
```

Abb. 4-10: Einige Standard-Fehlermeldungen

Ist die Validierung etwas komplexer oder muss auch semantisch validiert werden, kann eine FormBean die *validate()*-Methode überschreiben, die automatisch vom *ActionServlet* aufgerufen wird, nachdem die Properties gesetzt wurden, die deklarative Validierung beendet wurde und bevor die *execute()*-Methode ausgeführt wird.

```
ActionErrors validate(ActionMapping mapping,
                      HttpServletRequest req);
```

Abb. 4-11: Syntax der validate()-Methode

Wie Abb. 4-11 zeigt, gibt die *validate()*-Methode eine HashMap von Fehler-Objekten an den Main-Controller zurück. Diese Fehler-Objekte sind vom Typ *ActionError* und sind Klassen, die die Fehlermeldungen beinhalten, die dem Benutzer angezeigt werden sollen. Jede Fehlermeldung hat als Schlüssel das dazugehörende Attribut. Ist die Liste leer, sind keine Fehler aufgetreten. Der Main-Controller speichert die HashMap als Request-Attribut. So kann das <*html:errors*> Tag darauf zugreifen. Dann leitet er den Request wieder an das Formular zurück.

Es fehlt in Struts ein Interface, das implementiert werden kann um eigene Validatoren zu erstellen, somit wird das Framework dem Kriterium K 3.4-2 nicht gerecht.

4.5 Daten-Bindung

In Struts übernimmt die Aufgabe der CommandBean die *FormBean*. Somit übt diese zusätzlich zu ihrer Rolle als Model auch die Funktion der CommandBean aus. Sie wird an ein Formular gebunden und ihre Properties werden mit den Werten der Benutzereingabe befüllt. Das bedeutet, dass sowohl das Model als auch die

Command-Bean vom Framework und somit von der Servlet-API abhängig sind und das Kriterium K 3.5-1 nicht erfüllt wird.

Der Vorteil an diesem Ansatz der Daten-Bindung ist, dass *FormBeans* auch Properties zur Verfügung stellen können, die mit den Formular-Buttons korrespondieren. So können auch die aus Buttons resultierenden Parameter automatisch an die Bean gebunden werden. Das erleichtert die Überprüfung, welcher Button gedrückt wurde und minimiert dadurch Controller-Code.

Je nachdem, ob im Action-Mapping der *struts-config.xml* Datei eine *Action* mit einer *FormBean* assoziiert wird oder nicht, wird die *FormBean* an das Formular gebunden. Der Zusammenhang zwischen CommandBean und Sub-Controller wird im „Action-Mapping“ in der *struts-config.xml* Datei beschrieben. Abb. 4-12 zeigt noch einmal einen Auszug des „Action-Mappings“ (vergleiche Abschnitt 4.1.1, Abb. 4-3):

```
<form-beans>
  <form-bean name="scheduleItem" type=
              "eval.struts.ScheduleItem" dynamic="no"/>
</form-beans>
<action-mappings>
  <action   name="scheduleItem"
            type="eval.struts.AddToScheduleAction"
            validate="true" input="/ScheduleEntryView.jsp"
            scope="session" path="/addToSchedule">
    <forward name="success" path="/viewSchedule.do" />
    <forward name="error" path="/ScheduleEntryView.jsp" />
  </action>
</action-mappings>
```

Abb. 4-12: Auszug aus struts-config.xml

Wie in Abb. 4-12 gezeigt wird, ist in der Beispiel-Applikation der Sub-Controller *AddToScheduleAction* mit der CommandBean *ScheduleItem* verknüpft. Dieser Zusammenhang wird durch das *name* Attribut ausgedrückt.

Die Bindung und die eventuell notwendige Typkonvertierung geschehen analog zu der JSP-Vorgangsweise, wenn das <*jsp:set property*>-Tag verwendet wird (vergleiche Abschnitt 3.5.) Die Kriterien K 3.5-2 und K 3.4-3 werden somit nicht erfüllt. Das fertige *ActionForm*-Objekt wird an die *execute()*-Methode des Sub-Controllers übergeben, damit die vom Request eingegangenen Werte dem System zugänglich gemacht werden.

Der vom Benutzer falsch eingegebene Wert kann nicht in der *FormBean* gespeichert werden, außer es handelt sich um einen String-Wert. Da aber auch im *ActionErrors*-

Objekt kein Platz ist, kann der falsche Wert dem Benutzer bei der wiederholten Anzeige des Formulars nicht angezeigt werden [John03]. Kriterium 3.5-4 wird deshalb nicht erfüllt.

4.6 Internationalisierung

Eine Web-Applikation ist „internationalisiert“, wenn ihre Seiten in der Sprache angezeigt werden, die der Ländereinstellung des Browsers entspricht. Struts unterstützt diese Anforderung, indem Bezeichnungen und Fehlermeldungen in so genannte Property-Dateien ausgelagert werden können, wodurch die zwei Kriterien K 3.6-2 und K 3.6-3 erfüllt werden.

Es wird eine Standard-Property-Datei angelegt, in der die Beschriftungen für die Standard-Sprache hinterlegt sind, z.B. *schedule.properties* für Englisch in der Beispiel-Applikation. Für jede Sprache, die von der Applikation unterstützt werden soll, gibt es eine eigene Datei, die den Namen der Standard-Property-Datei um ein Sprachen- und/oder Länderkürzel erweitert (wie zum Beispiel *schedule_de.properties* für Deutsch oder *schedule_de_at.properties* für Deutsch in Österreich). So können die Web-Seiten gemäß der geforderten Sprache dargestellt werden.

Struts erhält vom Browser die Ortsinformation und verwendet zum Aufbau der Webseite die Beschriftungen und Titel der entsprechenden Property-Datei (Erfüllung des Kriteriums K 3.6-1). Ist für das angegebene Land keine eigene Datei vorhanden (z.B. für Deutschland *schedule_de_de.properties*), wird die Standard-Datei dieser Sprache verwendet (z.B. *schedule_de.properties*). Gibt es auch keine Datei für die angeforderte Sprache, so wird die Webseite in der Standard-Sprache angezeigt (z.B. *schedule.properties*) [Ford03].

Abb. 4-13 zeigt einen Ausschnitt aus der Standard-Property-Datei der Beispiel-Applikation:

```
prompt.duration=Duration
prompt.eventType=Event Type
prompt.start=Start Date
prompt.text=Text
[…]
```

Abb. 4-13: Auszug aus schedule.properties

Die in der Property-Datei hinterlegten Bezeichnungen werden sowohl für die Validierung (vergleiche Abschnitt 4.4, Abb. 4-9) als auch in der View (vergleiche Abschnitt 4.3, Abb. 4-6) verwendet.

4.7 Anwendungsspezifisches

Im diesem Abschnitt werden nur die Funktionalitäten beschrieben, die durch die Anwendungs-spezifischen Kriterien gefordert werden und auch vom Framework unterstützt werden.

4.7.1 Datei-Upload-Unterstützung

Struts erleichtert den Datei-Upload, indem das Custom-Tag <*html:file*> verwendet wird um das Formular zu erstellen. Ein Attribut der FormBean korrespondiert mit dem <*html:file*>-Element und stellt eine Änderungsmethode für dieses Attribut zur Verfügung. Dadurch kann das Attribut automatisch gesetzt werden und die Datei wird auf den Server geladen. Abb. 4-14 zeigt einen Ausschnitt eines solchen Formulars:

```
<html:form  action="uploadAction.do"
            enctype="multipart/form-data">
  Please Input Text:
  <html:text property="myText"/>
  Please Input The File to Upload:
  <html:file property="myFile"/>
  <html:submit />
</html:form>
```

Abb. 4-14: Verwendung des Struts Custom-Tags <html:file>

Wie in Abb. 4-14 ersichtlich ist, wird in dem Formular das Custom-Tag <*html:file*> verwendet. Das Attribut *myFile* korrespondiert mit der *property*-Ausprägung im <*html:file*>-Element.

Abb. 4-15 zeigt die zum oben gezeigten Formular gehörende *FormBean*:

```
public class UploadForm extends ActionForm {
    protected String myText;
    protected FormFile myFile;
    […]
    public void setMyFile(FormFile file) {
        myFile = file;
    }
    public FormFile getMyFile() {
        return myFile;
    }
}
```

Abb. 4-15: Beispiel für eine Form-Bean, die einen Datei-Upload beinhaltet

Die Datei wird auf den Server geladen und es kann mittels des Attributs *myFile* auf die Datei zugegriffen werden. Abb. 4-16 zeigt, wie zum Beispiel in der *execute()*-Methode des Sub-Controllers auf die Datei zugegriffen wird.

```
  ((UploadForm) form).getMyFile()
```

Abb. 4-16: Teil der execute()-Methode, der auf den Upload zugreift

4.8 Dokumentation

Die Referenz-Dokumentation von Struts ist aktuell und umfassend (K 3.8-1). Negativ erscheint einzig die Strukturierung, die die Dokumentation unübersichtlich erscheinen lässt. Es ist leider nicht offensichtlich zu erkennen, wo die gewünschte Information eingeordnet worden ist. Die JavaDocs sind ausführlich kommentiert und stellen eine gute Programmier-Hilfe dar (K 3.8-2). Leider gibt es keinen direkten Link von der Referenz-Dokumentation zu den JavaDocs.

Die große Beliebtheit von Struts macht sich natürlich auch in der Literatur bemerkbar. Im Internet können viele Artikel über Struts und seine Verwendung gefunden werden. Auf der Seite *http://struts.apache.org/faqs/index.html* werden FAQs und HowTos angeboten. Weiters gibt es eine Wiki-Seite, die unter *http://wiki.apache.org/struts* zu finden ist, wo „Best Practices“ preisgegeben werden und Tipps und Tricks zu finden sind. Neben [Hust02] ist noch eine Reihe anderer Bücher über Struts erhältlich, von denen einige unter *http://wiki.apache.org/struts/StrutsBooks* aufgelistet sind (K3.8-3).

4.9 Zusammenfassung

Folgende Tabelle soll zum Abschluss der Beschreibung des Struts-Frameworks eine zusammenfassende Übersicht über die Eigenschaften von Struts in Bezug auf die Evaluierungskriterien geben:

Controller	Sub-Controller sind JavaBeans	(K 3.1-1)	✓
	Superklassen für zwei verschiedene Kontrollflüsse	(K 3.1-2)	×
	Controller enthalten keine Datenbank-Informationen	(K 3.1-3)	×
Model	Model ist Servlet- und Framework-unabhängig	(K 3.2-1)	×
	Model kann mehrere JavaBeans enthalten	(K 3.2-2)	✓
	Model ist durch eigene Komponente realisiert	(K 3.2-3)	✓
View	Verschiedene View-Technologien werden unterstützt	(K 3.3-1)	✓
	View-Austausch ist ohne Controller-Änderung möglich	(K 3.3-2)	✓
Formular-Validierung	Validatoren sind Web-unabhängige JavaBeans	(K 3.4-1)	×
	Es können eigene Validatoren implementiert werden	(K 3.4-2)	×
	Deklarative Validierung ist möglich	(K 3.4-3)	✓
	Fehlermeldungen können angezeigt werden	(K 3.4-4)	✓
Daten-Bindung	CommandBean ist Servlet- und Framework-unabh.	(K 3.5-1)	×
	Typüberprüfungen und Typkonvertierungen	(K 3.5-2)	×
	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	(K 3.4-3)	×
	Falsche Daten werden im Formular angezeigt	(K 3.4-4)	×
Internationalisierung	Länder-Information wird zur Verfügung gestellt	(K 3.6-1)	✓
	Beschriftungen sind vom Java-Code getrennt	(K 3.6-2)	✓
	Fehlermeldungen sind vom Java-Code getrennt	(K 3.6-3)	✓
Anwendungs-spezifisches	JDBC-Abstraktions-Mechanismus	(K 3.8-1)	×
	Deklarative Transaktionsverwaltung ist möglich	(K 3.8-2)	×
	Verwaltung der Abhängigkeiten	(K 3.8-3)	×
	Wizard-Unterstützung	(K 3.8-4)	×
	Datei-Upload-Unterstützung	(K 3.8-5)	✓
Doku-mentation	Referenz-Dokumentation	(K 3.7-1)	✓
	JavaDocs	(K 3.7-2)	✓
	Beispielanwendungen und Tutorials	(K 3.7-3)	✓
Ergebnis	Erfüllte Kriterien:	**14 von 27**	

Tabelle 1: Struts in Bezug auf die Evaluierungskriterien

5. Kapitel

Spring

Das Spring-Projekt umfasst nicht nur ein aktionsgesteuertes Web-Framework, sondern auch ein Framework für die Geschäftslogik und eines für die Datenbankzugriffslogik. Das Projekt ist mittlerweile in der Version 1.1 verfügbar [Spri04]. In Abb. 5-1 wird der aktionsgesteuerte Ablauf und das Zusammenspiel der dabei beteiligten Spring-Komponenten dargestellt.

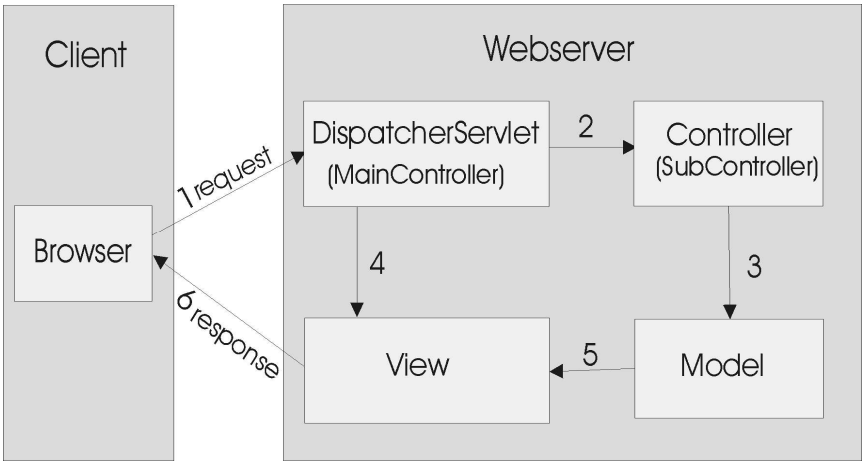

Abb. 5-1: Zusammenspiel der Spring-Komponenten

Spring's Web-Framework ist rund um die generische Framework-Klasse *Dispatcher Servlet* aufgebaut, die durch Konfigurationsdateien parametrisiert wird und den Main-Controller darstellt. Das *DispatcherServlet* behandelt die eingehenden Requests (1) und leitet sie an den entsprechenden (Sub)-*Controller* weiter (2). Dieser befüllt das Model (3), das die Daten für die Anzeige zur Verfügung stellt und gibt die Kontrolle wieder an das *DispatcherServlet* zurück. Von diesem wird dann die richtige View

ausgewählt (4). Mit Hilfe der Model-Daten wird die View generiert (5) und schließlich im Browser dargestellt (6).

5.1 Controller

Wie auch in allen anderen Frameworks, die in dieser Arbeit beschrieben werden, übernimmt der Sub-Controller die nicht generischen Aufgaben des Main-Controllers, deshalb gliedert sich dieses Kapitel in zwei Abschnitte – Main-Controller und Sub-Controller.

5.1.1 Main-Controller (DispatcherServlet)

Der Main-Controller von Spring, das *DispatcherServlet*, wird, wie auch alle anderen Servlets, im Deployment-Descriptor (*web.xml*) der Web-Applikation definiert. Es handelt sich dabei um eine konkrete Klasse, die nicht mehr erweitert werden muss. Abb. 5-2 zeigt die Deklaration des Main-Controllers:

```
<context-param>
   <param-name>contextConfigLocation</param-name>
  <param-value>
     /WEB-INF/applicationContext.xml
   </param-value>
</context-param>
<servlet>
   <servlet-name>spring</servlet-name>
   <servlet-class>
     org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>spring</servlet-name>
   <url-pattern>spring/*.do</url-pattern>
</servlet-mapping>
```

Abb. 5-2: Auszug aus web.xml

Abb. 5-2 zeigt, dass im Abschnitt <*servlet*> sowohl der Name als auch die vom Framework gelieferte Klasse, die als Main-Controller eingesetzt wird, deklariert wird. Innerhalb des <*servlet-mapping*>-Bereichs folgt dem zuvor definierten Servlet-Namen die Angabe eines URL-Musters, das den Pfad beschreibt, mit dem der Main-Controller erreicht werden kann. In dem <*context-param*>-Element wird eine Konfigurationsdatei *(applicationcontext.xml)* spezifiziert, die nicht für die Parametrisierung des Servlets verwendet wird, sondern für die Konfiguration der

Geschäftslogik. Auf die Funktion dieser Datei wird in Abschnitt 5.7 genauer eingegangen.

Bei der Initialisierung des Main-Controllers wird im WEB-INF-Verzeichnis nach einer *[servlet-name]-servlet.xml* Datei gesucht. Diese Datei dient dazu, die Komponenten des Web-Frameworks zu konfigurieren. Abb. 5-3 zeigt einen Ausschnitt der Konfigurationsdatei für die Beispiel-Applikation, die in dieser Evaluierung als grundlegendes Beispiel dient (vergleiche Abschnitt 2.3):

```
<bean  id="viewScheduleController"
       name="/viewSchedule.do"
       class="eval.spring.ViewScheduleController">
  <property name="sdb">
    <ref bean="scheduleDb"/>
  </property>
</bean>
<bean  id="addToScheduleController"
       name="/addToSchedule.do"
       class="eval.spring.AddToScheduleController">
  <property name="sdb">
    <ref bean= "scheduleDb"></ref>
  </property>
  <property name="validator">
    <ref bean= "siValidator"></ref>
  </property>
</bean>
```

Abb. 5-3: Auszug aus spring-servlet.xml

Wie in Abb. 5-3 zu sehen ist, werden in der *spring-servlet.xml*-Datei der Beispiel-Applikation zwei JavaBeans als Sub-Controller definiert. Spring erfüllt somit Kriterium K 3.1-1. *ViewScheduleController* ist für die Anzeige des Zeitplans zuständig und *AddToScheduleController* übernimmt sowohl die Formularanzeige als auch seine Verarbeitung. Das ist möglich, da sich in Spring der Kontrollfluss einer Formularseite von dem einer Standard-Seite unterscheidet, damit wird das Kriterium K 3.1-2 erfüllt.

Das Attribut *id* dient dazu, die Sub-Controller auch von einer anderen Klasse referenzieren zu können. Mit *name* wird der Pfad angegeben, über den sie erreichet werden können und durch das *class*-Attribut werden ihnen die implementierenden Klassen zugeordnet.

In Spring können die Abhängigkeiten von Klassen außerhalb des Java-Codes verwaltet werden. Das geschieht dadurch, dass nicht die Klasse selbst ihre Felder instanziert, sondern diese Aufgabe das Framework übernimmt. Das hat den Vorteil, dass die konkreten Implementierungen eines Typs jederzeit ausgetauscht werden können, ohne Änderungen im Code vornehmen zu müssen. Stimmt das *name*-Attribut eines <*property*>-Elements mit einem Property der Sub-Controller-Klasse überein, wird das Feld vom Framework automatisch gesetzt.

Im Beispiel hat der *ViewScheduleController* ein Attribut namens *sdb*, das vom Typ *ScheduleDb* ist (siehe Abb. 5-4.). Durch die oben gezeigte <*ref bean*> Deklaration wird ein *ScheduleDb*-Objekt instanziert und dem Attribut *sdb* zugewiesen. Für genauere Informationen zur externen Verwaltung von Abhängigkeiten siehe Abschnitt 5.7.3.

5.1.2 Sub-Controller (Controller)

Wie im „Model 2“-Entwurfsmuster beschrieben ist, interpretieren Sub-Controller Benutzereingaben und erzeugen daraus ein Model, das dann von der View dargestellt wird. In Spring wird ein Sub-Controller einfach Controller genannt, deshalb wird auch in diesem Kapitel dieser Ausdruck verwendet. Die Basis der Controller-Hierarchie ist das einfache Interface *Controller*, das nur die Methode

```
ModelAndView handleRequest request, response)
```

zu Verfügung stellt. Es beinhaltet die wichtigste Funktionalität, die allen Controllern gemeinsam ist, nämlich die Bearbeitung eines Requests und die Rückgabe von Model und View. Das *ModelAndView*-Objekt enthält eine Map, die das Model darstellt (siehe 5.2) und einen String, mit dem die anzuzeigende View (siehe 5.3) ermittelt werden kann.

Spring stellt eine ganze Reihe von Implementierungen dieses Interfaces zur Verfügung, die verschiedenste Funktionalitäten bieten. Es kann die Sub-Controller-Klasse abgeleitet werden, dessen Verhalten gerade benötigt wird. Hat man zum Beispiel kein Formular, benötigt man auch keinen Formular-Controller. Die zur Verfügung gestellten Controller können in drei Gruppen aufgeteilt werden: Einfache Controller, Multi-Action-Controller und Command-Controller.

Einfache Controller eignen sich für Aktionen, bei denen keine Daten-Bindung notwendig ist. Zum Beispiel für Seiten, die keine Benutzerdaten aufnehmen sondern nur Daten anzeigen. Als fertige Implementierung stellt Spring den *ParameterizableViewController* zur Verfügung. Bei der Verwendung dieses Controllers kann die anzuzeigende View in der *[servlet-name]-servlet.xml*-Datei definiert werden, was die Erfüllung des Kriteriums K 3.3-3 bedeutet, weil dadurch die View ohne Änderung am Controller-Code ausgetauscht werden kann. In der Beispiel-Applikation kann ein solcher Controller für die erste Aufgabe erweitert werden, die nur darin besteht die in der Datenbank vorhandenen Termine anzuzeigen. Abb. 5-4 zeigt die Verwendung dieses Controllers:

```
public class ViewScheduleController extends
                    ParameterizableViewController{

    private ScheduleDb sdb;

    public void setSdb(ScheduleDb sdb) {
        this.sdb = sdb;
    }

    protected ModelAndView handleRequestInternal(
                    HttpServletRequest request,
                    HttpServletResponse response) {
        Map model= new HashMap();
        List events= sdb.getEvents();
        model.put("events", events);
        return new ModelAndView(getViewName(),model);
    }
}
```

Abb. 5-4: Auszug aus ViewScheduleController.java

Wie in der *spring-servlet.xml*-Datei (Abb. 5-3) weiter oben zu sehen war, hat der *ViewScheduleController* ein Property namens *sdb*. Die Controller-Klasse stellt eine entsprechende Änderungs-Methode zur Verfügung, so kann das Attribut *sdb* automatisch gesetzt werden, sofern die Klasse *ScheduleDb* einen parameterlosen Standard-Konstruktor bietet, und muss nicht in der Controller-Klasse angelegt werden.

Die anzuzeigenden Termine werden mittels der Methode *getEvents()* des *ScheduleDb*-Objekts aus der Datenbank geholt und in eine HashMap eingefügt, die das Model (siehe 5.2) darstellt. Der View-Name wird in der *spring-servlet.xml*-Datei als Property des Controllers konfiguriert und kann mit *getViewName()* dem *ModelAndView*-Objekt übergeben werden.

Multi-Action-Controller stellen einen anderen Kontrollfluss als die übrigen Controller dar. Dabei wird nicht jeder Aktion ein Controller zugeordnet, sondern jede Aktion korrespondiert mit einer Methode im Controller. Diese Zuordnung geschieht mit Hilfe eines *MultiactionMethodNameResolvers*, für den Spring mehrere Implementierungen zur Verfügung stellt. Jede Methode in der *MultiActionController*-Subklasse muss folgende Signatur haben, damit sie als Action-Methode erkannt wird.

```
ModelAndView anyMethodName(HttpServletRequest,
                  HttpServletResponse, ExceptionClass)
```

Abb. 5-5: Methoden-Signatur einer MultiActionController-Action

Die *ExceptionClass* kann jede beliebige Unterklasse von *java.lang.Exception* oder *java.lang.RuntimException* sein. Multi-Action-Controller eignen sich für die Fälle, in denen viele einfache Controller benötigt werden. So muss nicht für jede Action ein neuer Controller implementiert werden, es genügt jeweils eine Methode zu schreiben. Auch in Situationen, in denen ein großes Maß an Funktionalität mehreren Controllern gemeinsam ist und trotzdem verschiedene Eintrittspunkte benötigt werden, ist ein Controller dieser Art hilfreich.

Command-Controller stellen die wichtigste der drei Gruppen dar, denn sie werden dann gebraucht, wenn Request-Parameter an Daten-Objekte gebunden werden sollen. Die zwei für die Implementierungen heranzuziehenden Superklassen sind *SimpleFormController* und *AbstractWizardFormController*. Beide binden Request-Parameter an das Objekt, das als CommandBean spezifiziert wird, bieten Validierung an und übernehmen den Kontrollfluss des Formulars. Je nachdem, ob die Datenmenge das Maß, das in einer Formularseite angezeigt werden kann, überschreitet oder nicht, wird der erste oder der zweite oben genannte Controller abgeleitet. In der Beispiel-Applikation liegt ein einseitiges Formular vor, deshalb wird *SimpleFormController* erweitert. Abb. 5-6 zeigt den *AddToScheduleController* der Beispiel-Applikation:

```
public class AddToScheduleController extends
                                SimpleFormController{
  private ScheduleDb sdb;
  private ScheduleItemValidator val;
  public void setVal(ScheduleItemValidator val) {
    this.val= val;
  }
  public void setSdb(ScheduleDb sdb) {
     this.sdb = sdb;
  }
…
```

```
 public AddToScheduleController() {
    setCommandName("scheduleItem");
    setCommandClass(ScheduleItem.class);
  }
  protected Map referenceData(HttpServletRequest request)
                              throws Exception {
      Map model= new HashMap();
      List events= sdb.getEvents();
      model.put("events", events);
      return model;
  }
  protected ModelAndView onSubmit(HttpServletRequest
                  request,HttpServletResponse response,
                  Object o, BindException e)
                  throws Exception {
      ScheduleItem si= (ScheduleItem)o;
      sdb.addRecord(si);
      response.sendRedirect("viewSchedule.do");
      return null;
  }
}
```

Abb. 5-6: Auszug aus AddToScheduleController.java

In Abb. 5-6 wird gezeigt, wie die von Spring bereitgestellten *FormController*-Implementierungen den Kontrollfluss eines Formulars übernehmen. Sie bieten eine Methode *referenceData()* an, mit der das Model der View in Form einer Map zugänglich gemacht wird, sodass beim erstmaligen Anzeigen des Formulars schon die notwendigen Informationen zur Verfügung stehen. In der Beispiel-Applikation werden so der View die anzuzeigenden Termine verfügbar gemacht.

Im Konstruktor des Controllers wird die CommandBean mit Namen und Klassenangabe spezifiziert, dadurch wird automatisch ein neues Objekt der angegebenen Klasse instanziert und an das Formular gebunden. Möchte man ein schon bestehendes Objekt binden, so kann man das mit folgender Methode erfolgen:

```
protected Object formBackingObject(HttpServletRequest req)
```

Abb. 5-7: Methode zur CommandBean-Bestimmung

Im obigen Beispiel aus Abb. 5-6 ist die CommandBean ein *ScheduleItem*-Objekt und wird mittels dem Standard-Konstruktor, der vorhanden sein muss, angelegt.

In der *onSubmit()*-Methode der Beispiel-Applikation wird die CommandBean einem *ScheduleItem*-Objekt zugewiesen und mittels der *addRecord()*-Methode der *ScheduleDb*-Klasse in die Datenbank gespeichert. Mit einem direkten

response.sendRedirect() wird der Request an den *ViewScheduleController* (siehe Mapping in der *spring-servlet.xml*-Datei) weitergeleitet.

Aus der Abb. 5-6 ist ersichtlich, dass der Controller keinerlei Datenbank bezogene Daten enthält. Das *ScheduleDb*-Objekt wird nämlich nicht im Controller angelegt, sondern vom Framework instanziert und durch die Angaben in der Konfigurationsdatei *applicationcontext.xml* wird die Datenquelle im Objekt gesetzt wodurch auch das Kriterium K 3.1-3 erfüllt ist.

5.2 Model (HashMap)

Ein Model besteht aus einem oder mehreren Java-Objekten, meistens JavaBeans. Jedes Model-Objekt hat einen Namen, mit dem es von der View angesprochen wird. Deshalb wird in Spring das Model als Map weitergegeben, auch wenn diese Map nur einen Eintrag haben wird. Dadurch ist das Model eine eigene Komponente und das Kriterium K 3.2-2 erfüllt. Außerdem ist das Model nicht auf ein Objekt beschränkt, wodurch der View Daten aus beliebigen Objekten zur Verfügung gestellt werden können. Es wird daher auch Kriterium 3.2-3 erfüllt. Die Rückgabe als Map hat den Vorteil, das das Model Servlet-API unabhängig ist, wodurch das Kriterium K 3.2-1 erfüllt wird, und keine Einschränkungen der verwendbaren Views daraus resultieren.

5.3 View

In Spring können Model-Daten in einem Web-Browser veröffentlicht werden ohne an eine bestimmte View-Technologie gebunden zu sein. Kriterium K 3.3-1 wird dadurch erfüllt. Es können JSP [Sun03a], Velocity [Velo04], XSLT [W3C99] und andere Technologien verwendet werden.

Der View-Resolver weist jedem View-Namen eine tatsächliche View zu. Weder die Klasse *ViewResolver* noch die Klasse *View* muss implementiert oder abgeleitet werden, da Spring schon fertige Klassen liefert, aus denen nur die gewünschte ausgesucht werden muss.

5.4 Formular-Validierung

Das von Spring verwendete Validierungskonzept ist wie folgt aufgebaut: Jedem Formular-Controller wird ein Validator zugeordnet. Validatoren sind Java-Klassen, die Benutzereingaben auf ihre Gültigkeit überprüfen. Abb. 5-8 zeigt die zwei Interfaces, die ein Validator implementieren muss:

```
public interface Validator {
   boolean support(Class c);
   void validate(Object o, Errors e);
}
public interface Errors{
   void rejectValue(String field, String code,
                    String message);
   […]
}
```

Abb. 5-8: Interfaces der Validatoren

Der Validator muss in der *support()*-Methode definieren, für welche Klassen er als Validator fungieren kann. Während er in der *validate()*-Methode die Überprüfungen durchführt. Diese Methode wird automatisch und noch vor der *onSubmit()*-Methode des Controllers aufgerufen. Gibt es Fehler, so fügt er sie dem *Errors*-Objekt mit Hilfe der *rejectValue()*-Methode hinzu. Auf dieses *Errors*-Object kann in der View zugegriffen werden. So wird Spring dem Kriterium K 3.4-4 gerecht.

Durch die Verwendung der in Abb. 5-8 gezeigten Interfaces sind die Validatoren Framework- und Servlet-API-unabhängig, sie erfüllen dadurch Kriterium K 3.4-1. Außerdem können dadurch Validatoren für beliebige eigene Objekte implementiert werden - Kriterium 3.4-2 wird erfüllt.

Der Validator ist ein Property (*validator*), das der Controller von der Spring-Oberklasse erbt. Das heißt er wird in der *applicationcontext.xml*-Datei definiert (siehe 5.7.3) und von der Controller-Deklaration in der *[servlet name]-servlet.xml* Datei referenziert (vergleiche Abb. 5-3). So kann er jederzeit gegen einen anderen ausgewechselt werden, sollte sich an den Validierungsregeln etwas ändern. Durch die Verwendung von XML-Dateien zur Konfiguration ist es möglich die Validatoren zu parametrisieren und so die Werte, nach denen validiert werden soll, flexibel zu halten. Somit bietet Spring nur in geringem Maße deklarative Validierung, – Kriterium K 3.4-3 wird nicht erfüllt – weil die Validatoren noch selbst geschrieben werden müssen.

5.5 Daten-Bindung

Durch die Spezifikation der Properties *commandName* und *commandClass*, die jeder Formular-Controller von Spring aufweist, wird die CommandBean mit dem Formular in Zusammenhang gebracht. Abb. 5-9 zeigt die CommandBean der Beispiel-Applikation:

```
public class ScheduleItem {
   private String start;
   private int duration;
   private String text;
   private String eventType;

   public ScheduleItem() {}
   public String getStart() {return start;}
   public void setStart(String start) {this.start = start;}
   […]
}
```

Abb. 5-9: Auszug aus ScheduleItem.java

Die in Abb. 5-9 gezeigte CommandBean der Beispiel-Applikation besteht aus vier Attributen mit deren Zugriffs- und Änderungs-Methoden. Im Unterschied zur *ScheduleItem*-Klasse in Struts, ist diese Bean vom Framework unabhängig, denn in Spring ist es nicht notwendig, Framework-Klassen abzuleiten um Daten-Bindung zu ermöglichen, was das Kriterium K 3.5-1 erfüllt.

Mittels der Daten-Bindung werden die Attribute der CommandBean automatisch mit den Werten der Request-Parameter, die den Formularfeldern entsprechen, befüllt. Spring verwendet im Gegensatz zu Struts nicht das *BeanUtil*-Package von Apache, sondert liefert eigene Bean-Manipulierung-Klassen. Spring übernimmt bei der Zuweisung von Parameter-Werten auf Bean-Properties die Typüberprüfungen und führt notwendige Konvertierungen bei Standardtypen durch. Kriterium K 3.5-2 wird dadurch erfüllt. Bei Typunverträglichkeiten werden die Ausnahmen und die betroffenen Felder mitsamt der Benutzereingabe in ein *Error*-Objekt geschrieben (vergleiche Abb. 5-8). Somit erfüllt diese Daten-Bindungs-Methode nicht nur das Kriterium K 3.5-3 sondern auch K 3.5-4, da bei der erneuten Anzeige des Formulars die zuvor eingegebenen Daten wieder angezeigt werden können.

In der View kann ein Formular-Feld mit dem Custom-Tag <*spring:bind*> an ein Property der Bean gebunden werden, um die Daten-Bindung auch in die andere Richtung zu unterstützen. Denn es wird so auch das Formularfeld automatisch mit

dem Wert des Bean-Attributs befüllt. Das bedeutet, dass das Feld bei erstmaliger Anzeige leer ist und bei wiederholter Darstellung den gespeicherten Wert anzeigt. Abb. 5-10 zeigt die Verwendung dieses Tags an Hand der Formularseite der Beispiel-Applikation:

```
<spring:bind path="duration">
  <input type="text" name="${status.expression}"
         value="<c:out value=”${status.value}”>" />
  <c:if test=”${status.error}”>
    <c:out value=“${status.errorMessage}”/>
  </c:if>
</spring:bind>
```

Abb. 5-10: Auszug aus addToSchedule.jsp

In Abb. 5-10 ist ersichtlich, wie in der Formularseite der Beispiel-Applikation das Attribut der CommandBean *duration* (vergleiche Abb. 5-9) mit einem Formularfeld in Zusammenhang gebracht wird. Bei der Verwendung des <*spring:bind*>-Tags ist es notwendig, mittels des Properties *path* anzugeben, mit welchem Attribut der CommandBean dieses Feld verknüpft werden soll. Die *status*-Variable dieses Tags enthält alle Informationen über das verbundene Attribut. So kann mit *status.expression* der Feldname angepasst werden, was dann wichtig ist, wenn die Seite in verschiedenen Sprachen angezeigt werden soll. Und mit *status.value* kann der Wert des Attributs ermittelt werden. Der Ausdruck *status.error* wird dazu verwendet, das Feld auf Fehler zu überprüfen und *status.errorMessage* dazu, um die Fehler anzuzeigen.

5.6 Internationalisierung

Spring bietet verschiedene Möglichkeiten an, die Sprach- und Länderinformation des Benutzers/ der Benutzerin heranzukommen. Die Länder- und Sprachinformation wird entweder aus dem Header des Requests, aus einem Cookie oder der Session ausgelesen. Mittels *RequestContext.getLocale()* können dann jederzeit diese Informationen abgefragt werden. Kriterium K 3.6-1 wird dadurch erfüllt.

In der *[servlet-name]-servlet.xml* Datei kann eine Property-Datei definiert werden, Abb. 5-11 zeigt eine solche Definition:

```
<bean  id="messageSource"
       class="org.springframework.context.support.
                          ResourceBundleMessageSource">
  <property name="basename">
    <value>messages</value>
  </property>
</bean>
```

Abb. 5-11: Auszug aus spring-servlet.xml

Der Abb. 5-11 kann entnommen werden, wie eine Property-Datei definiert wird. In Spring heißt eine Property-Datei „MessageSource“. Diese ist im „WEB-INF/classes“-Ordner gespeichert und beinhaltet beliebige Elemente, so auch Titel und Feldbeschriftungen, die dann je nach Land und/oder Sprache angezeigt werden können, was die Erfüllung des Kriteriums K 3.6-2 bedeutet.

Es muss für die Deklaration der Basisname als Property der „MessageSource“ angegeben werden, im obigen Beispiel ist dieser *messages*. An diesen Basisnamen können dann die Standard Länder- und Sprachkürzel angehängt werden. Zum Beispiel *_de* für die Sprache Deutsch und *_at* für das Land Österreich, es entsteht so der Dateiname *messages_de_at.properties*. Diese Datei wird dann herangezogen, wenn die Sprach- und Länderinformation Deutsch und Österreich ergibt.

Mit Hilfe des Tags <*spring:message*> können die Elemente der Property-Datei angesprochen und in der Web-Seite verwendet werden. Wie im Kapitel 5.4 schon gezeigt wurde, sind die Fehlermeldungen in einem eigenen *Error*-Objekt vom Validierungscode getrennt. Dadurch ist auch die Anpassung dieser Meldungen an verschiedene Sprachen möglich und Kriterium K 3.6-3 wird erfüllt.

5.7 Anwendungsspezifisches

In diesem Abschnitt werden nur die Funktionalitäten beschrieben, die durch die Anwendungs-spezifischen Kriterien gefordert werden und auch vom Framework unterstützt werden.

5.7.1 JDBC-Abstraktion

Spring stellt ein eigenes JDBC-Abstraktions-Framework zur Verfügung. Somit muss nicht direkt mit der Java-JDBC-API gearbeitet werden, sondern der Datenbankzugriff wird mit wieder verwendbaren Java-Klassen durchgeführt.

Die Klasse *JdbcDaoSupport* kann für die Service-Klassen, die den Datenbankzugriff kapseln, als Oberklasse dienen, sie bietet ein Property *dataSource* mit entsprechender Änderungs- und Zugriffsmethode. Mit Hilfe dieser Methoden kann die Datenquelle in der *applicationcontext.xml*-Datei konfiguriert werden und automatisch gesetzt werden (siehe Kapitel 5.7.3). Abb. 5-12 stellt das Datenbank-Service *ScheduleDb* der Beispiel-Applikation dar:

```
public class ScheduleDb extends JdbcDaoSupport {
   private InsertScheduleItem insSi;
   private GetAllEvents getEvents;
   public ScheduleDb() {
      insSi= new InsertScheduleItem(getDataSource());
      getEvents= new GetAllEvents(getDataSource());
   }
   public void addRecord(ScheduleItem si){
      insSi.doInsert(si);
   }
   public List getEvents(){
      return getEvents.execute();
   }
   […] //private classes
}
```

Abb. 5-12: Auszug aus ScheduleDb.java

Die in Abb. 5-12 gezeigte Klasse *ScheduleDb* der Beispiel-Applikation enthält die Datenbankzugriffslogik; mit ihrer Methode *addRecord()* kann ein neues *ScheduleItem* (vergleiche Abb. 5-9) in die Datenbank gespeichert werden und mit *getEvents()* wird die Liste der bestehenden Termine aus der Datenbank geholt. Jede Datenbankoperation wird durch eine Java-Klasse, die eine Framework-Klasse erweitert, realisiert. In der Beispiel-Applikation werden für den Datenbankzugriff zwei Klassen *InsertScheduleItem* und *GetAllEvents* verwendet. Abb. 5-13 zeigt die erste der beiden:

```
private class InsertScheduleItem extends SqlUpdate{

  public InsertScheduleItem(DataSource dataSource) {
    super(dataSource, "insert into sched_events”+
                        +”values(?,?,?,?)");
    declareParameter(new SqlParameter(Types.VARCHAR));
    declareParameter(new SqlParameter(Types.VARCHAR));
    declareParameter(new SqlParameter(Types.NUMERIC));
    declareParameter(new SqlParameter(Types.VARCHAR));
    compile();
  } …
 public void doInsert(ScheduleItem si){
    update(new Object[]{si.getText(),si.getStart(),
      new Integer(si.getDuration()),si.getEventType()});
 }
}
```

Abb. 5-13: Klasse, die eine Insert-Operation durchführt

Abb. 5-13 zeigt eine Klasse, die *SqlUpdate* ableitet. Diese Klasse wird erweitert, wenn Insert-, Update- oder Delete-Statements auf die Datenbank ausgeführt werden. Im Konstruktor müssen die Datenquelle und die SQL-Anweisung gesetzt werden. Es wird daraus ein PreparedStatement erzeugt. Sind für die Anweisung Parameter nötig, wird für jedes Fragezeichen in der SQL-Anweisung der entsprechende Parametertyp mittels der Methode *declareParameter()* definiert. Der Methode *update()* werden alle Parameterwerte in Form eines Objekt-Arrays übergeben und somit wird die Datenbankoperation ausgeführt.

Wird eine Datenbankabfrage benötigt, wird die Klasse *MappingSqlQuery* erweitert. In Abb. 5-14 wird die zweite der Datenbankzugriffs-Klassen der Beispiel-Applikation *GetAllEvents* beschrieben, deren Aufgabe es ist die vorhandenen Termine aus der Datenbank auszulesen:

```
private class GetAllEvents extends MappingSqlQuery{
  public GetAllEvents(DataSource ds) {
    super(ds, "select * from sched_events");
  }
  protected Object mapRow(ResultSet rs, int i) throws
                                    SQLException {
    String text= rs.getString("text");
    String start= rs.getString("start");
    int duration= rs.getInt("duration");
    String et= rs.getString("eventtype");
    return new ScheduleItem(text, duration, start, et);
  }
}
```

Abb. 5-14: Klasse, die eine Abfrage realisiert

Im Konstruktor der in Abb. 5-14 gezeigten Klasse *GetAllEvents* wird die Datenquelle und die SQL-Anweisung gesetzt. Zusätzlich muss die Methode *mapRow()* implementiert werden. In dieser wird das gewünschte Objekt *ScheduleItem* aus dem ResultSet zusammengebaut und zurückgegeben. Die geerbte Methode *execute()*, die in der Klasse *ScheduleDb* aufgerufen wird (vergleiche Abb. 5-12), kann dadurch eine Liste mit *ScheduleItems* liefern.

5.7.2 Transaktions-Abstraktion

Transaktionen werden entweder global oder lokal verwendet. Globale Transaktionen werden vom Applikationsserver durch die Verwendung von JTA (Java Transaction API [sun00]) verwaltet. Eine beliebte Art, diese Transaktionsform zu verwenden ist mittels EJB, da der Container in diesem Fall die Transaktionsverwaltung und somit auch die aufwendige JTA-Verwendung übernimmt. Wird die Applikation jedoch ohne EJB implementiert steht diese Vereinfachungsmöglichkeit nicht zur Verfügung.

Lokale Transaktionen sind mit speziellen Ressourcen assoziiert, wie zum Beispiel einer JDBC-Verbindung. Ihr Nachteil liegt darin, dass Code, der lokal eine Transaktion bezüglich einer Ressource verwaltet, nicht in einer globalen Transaktion laufen kann.

Mit Spring können Transaktionen deklarativ beschrieben werden, oder durch Programmierung gehandhabt werden. Beide Möglichkeiten sind von der darunter liegenden Transaktions-API unabhängig und können somit in jeder Umgebung eingesetzt werden. Es wird jedoch von Spring die deklarative Form der Transaktionsverwaltung empfohlen, da sie einfacher zu verwenden ist und die gleichen Möglichkeiten bietet wie das Programmierung der Transaktionen.

Abb. 5-15 zeigt am Beispiel der Referenz-Implementierung wie in der Konfigurationsdatei eingestellt werden kann, dass die Methode *addRecord()* der Klasse *ScheduleDb* transaktional sein soll:

```
<bean id="transactionManager" class="org.springframework.
          jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource"><ref bean="dataSource"/>
   </property>
</bean>
…
```

```
…
<bean id="myManager" class="org.springframework.
    transaction.interceptor.TransactionProxyFactoryBean">
   <property name="transactionManager">
      <ref bean="transactionManager"/>
   </property>
   <property name="target">
      <ref bean="myManagerTarget"/>
   </property>
   <property name="transactionAttributes">
      <props>
        <prop key="addRecord">PROPAGATION_REQUIRED</prop>
      </props>
   </property>
</bean>
<bean  id="myManagerTarget"
       class="eval.spring.ScheduleDb"/> […]
```

Abb. 5-15: Auszug aus applicationcontext.xml

Zuerst wird in der *applicationcontext.xml*-Datei ein *TransaktionsManager* deklariert. Diese von Spring zur Verfügung gestellte Klasse benötigt die Datenquelle als Property.

Die Klasse ist das Ziel-Objekt, deren Methoden transaktional sind. Sie wird in ein Transaktions-Proxy-Objekt gehüllt. Dieses Proxy ist vom Typ *TransactionProxy FactoryBean* und benötigt eine Referenz zum Transaktions-Manager, zu seinem Ziel-Objekt und zu den Transaktions-Attributen.

In dem <*Props*>-Element werden die Methoden definiert, die transaktional sein sollen. In dem Beispiel aus Abb. 5-15 ist das die Methode *addRecord()*. Bei dieser Deklaration können verschiedene Transaktions-Definitionen angegeben werden, wie *timeout, isolation, propagation* oder *read-only-status.*

5.7.3 Verwaltung der Abhängigkeiten

Die Klasse A ist von der Klasse B abhängig, wenn sie ein Objekt der Klasse B instanziert, da falls sich an der Implementierung des B-Objektes etwas ändert, der Java-Code der A-Klasse ebenfalls geändert werden muss. In Spring können die Abhängigkeiten von Klassen außerhalb des Java-Codes verwaltet werden. Das wird dadurch realisiert, dass nicht die Klasse selbst ihre Felder instanziert, sondern diese Aufgabe das Framework übernimmt. Das hat den Vorteil, dass die konkreten Implementierungen eines Typs jederzeit ausgetauscht werden können, ohne

Änderungen im Code notwendig zu machen. Diese Auslagerung der Abhängigkeiten zwischen Beans nennt man „Inversion of Control“ [John03].

Es können nicht nur die Beziehungen von Klassen der Präsentationslogik in XML-Konfigurationsdateien ausgelagert werden, sondern auch die aller anderen Klassen. Es muss kein spezielles Interface implementiert werden, sondern die zu erzeugende Klasse muss einen parameterlosen Standard-Konstruktor aufweisen und die Verwender-Klasse muss entsprechende Änderungs-Methoden zur Verfügung stellen. Abb. 5-16 zeigt die *applicationcontext.xml*-Datei, die für die Konfiguration der Klassen der Geschäftslogik verwendet wird:

```
<bean id="dataSource"
      class="...commons.dbcp.BasicDataSource">
  […]
  <property name="username">
    <value>root</value>
  </property>
  <property name="password">
    <value>marathon</value>
  </property>
</bean>
<bean id="siValidator" class="ScheduleItemValidator">
  <property name=”min”><value>0</value></property>
  <property name=”max”><value>31</value></property>
</bean>
<bean id="scheduleDb" class="ScheduleDb">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

Abb. 5-16: Auszug aus applicationcontext.xml

Wie der Abb. 5-16 entnommen werden kann, wird in der *applicationcontext.xml*-Datei die Datenquelle definiert. So kann, ohne dass Änderungen im Java-Code notwendig sind, die Datenquelle ausgetauscht werden. Danach wird der Validator angegeben, der in der Applikation benötigt wird (vergleiche Kapitel 5.4.) Damit auch die *ScheduleDb*-Klasse vom Controller aus referenziert werden kann, (vergleiche Abb. 5-3) wird anschließend auch diese deklariert.

5.7.4 Wizard-Controller

Übersteigt die Anzahl der Eingabefelder das Kontingent, das eine Formularseite fassen kann, müssen die Formularinhalte auf mehrere Seiten aufgeteilt werden. Ist die

Verwendung solcher Wizards in einer Applikation notwendig, kann die Spring-Klasse *AbstractWizardController* an Stelle der *SimpleFormController*-Klasse abgeleitet werden. Abb. 5-17 zeigt den Konstruktor eines Wizard-Controllers:

```
public SomeWizardController() {
   setCommandName("…”);
   setCommandClass(Command.class);
   setPages(new String[]{"page0","page1",”page2”});
   setSessionForm(true);
}
```

Abb. 5-17: Konstruktor eines WizardControllers

Im Konstruktor werden neben dem Name der CommandBean und ihrer Klasse auch noch die Seiten angegeben, die im Zuge der Wizard-Formular-Eingabe angezeigt werden sollen. Die Methoden *referenceData()* und *onBindAndValidate()* unterscheiden sich von denen im *SimpleFormController* durch einen zusätzlichen Parameter *page* vom Typ int. Mittels diesem kann in den Methoden auf die gerade aktuelle Seite eingegangen werden. In diesem Controller wird nicht mehr automatisch die *validate()*-Methode des definierten Validators aufgerufen, sondern das muss in der *validate()*-Methode des Controllers, die in Abb. 5-18 gezeigt wird, erledigt werden.

```
protected void validatePage(Object command, Errors e, int
page)
```

Abb. 5-18: Validierungs-Methode des Wizard-Controllers

An Stelle von *onSubmit()* müssen zwei Methoden überschrieben werden, die in Abb. 5-19 dargestellt sind:

```
protected ModelAndView processFinish(
                  HttpServletRequest request,
                  HttpServletResponse response,
                  Object command, BindException errors)

protected ModelAndView processCancel(
                  HttpServletRequest request,
                  HttpServletResponse response,
                  Object command, BindException errors)
```

Abb. 5-19: Submit-Methoden des Wizard-Controllers

Die *Speichern* und *Abbrechen* Buttons der JSP-Seiten müssen *_finish* und *_cancel* heißen. Um auf die verschiedenen Seiten im Wizard zu verlinken muss der Name des Buttons *_targetX* sein, wobei X für die Index-Nummer im *pages*-Array des Controllers steht.

5.7.5 Datei-Upload-Unterstützung

Spring bietet auch Unterstützung für Formulare, mit denen Dateien auf den Server geladen werden können. Diese wird durch das Interface *MultipartResolver* realisiert. Abb. 5-20 zeigt die Deklaration eines im Framework enthaltenen *MultipartResolvers* in der Konfigurationsdatei von Spring:

```
<bean  id="multipartResolver"
       class="…web.multipart.cos.CosMultipartResolver">
```

Abb. 5-20: Auszug aus [servlet-name]-servlet.xml

Wird ein solcher *MultipartResolver* verwendet, wird jeder Request danach untersucht, ob mit ihm zusätzlich auch Dateien geladen werden sollen. Ist das der Fall, können diese vom angegebenen *MultipartResolver* verarbeitet werden. Anschließend können sie, wie jedes andere Attribut des Requests, abgefragt werden. Im Grunde ist die Vorgehensweise beim Datei-Upload folgende: das aktuellen *HttpServletRequest*-Objekt wird in ein *MultipartHttpServletRequest*-Objekt eingehüllt, das Dateien laden kann. Der Controller kann dadurch auf die im Request enthaltenen Dateien zugreifen und auch Informationen über diese abfragen. In Abb. 5-21 wird dargestellt, wie die geladenen Datei im Controller zugänglich ist:

```
MultipartHttpServletRequest multipartRequest =
                   (MultipartHttpServletRequest)request;
MultipartFile multipartFile =
                   multipartRequest.getFile("uploadfile");
```

Abb. 5-21: Auszug aus dem Upload-Controller

5.8 Dokumentation

Da Spring eine Weiterentwicklung des in dem Buch *J2EE Design and Development* von Rod Johnson [John03] vorgestellten Frameworks ist, informiert dieses Buch über die grundlegenden Funktionsweisen. In Rod Johnsons neuem Buch *J2EE without EJB* [John04] wird noch genauer und spezieller auf Spring eingegangen. Die Referenz-Dokumentation ist in manchen Bereichen sehr genau und beschreibt die internen Vorgänge detailliert, leider sind jedoch manche Kapitel nicht vollständig und es mangelt durchgehend an Beispielen zum besseren Verständnis der Anwendung. Kriterium K 3.8-1 ist nicht erfüllt. Die JavaDocs sind gut kommentiert und es findet sich Information sowohl auf Klassen als auch auf Methodenebene, was die Erfüllung des Kriteriums K 3.8-2 bedeutet. Das zur Verfügung gestellte Tutorial genügt dem Kriterium K 3.8-3.

5.9 Zusammenfassung

Folgende Tabelle soll zum Abschluss der Beschreibung des Spring-Frameworks eine zusammenfassende Übersicht über die Eigenschaften von Spring in Bezug auf die Evaluierungskriterien geben:

Controller	Sub-Controller sind JavaBeans	(K 3.1-1)	✓
	Superklassen für zwei verschiedene Kontrollflüsse	(K 3.1-2)	✓
	Controller enthalten keine Datenbank-Informationen	(K 3.1-3)	✓
Model	Model ist Servlet- und Framework-unabhängig	(K 3.2-1)	✓
	Model kann mehrere JavaBeans enthalten	(K 3.2-2)	✓
	Model ist durch eigene Komponente realisiert	(K 3.2-3)	✓
View	Verschiedene View-Technologien werden unterstützt	(K 3.3-1)	✓
	View-Austausch ist ohne Controller-Änderung möglich	(K 3.3-2)	✓
Formular-Validierung	Validatoren sind Web-unabhängige JavaBeans	(K 3.4-1)	✓
	Es können eigene Validatoren implementiert werden	(K 3.4-2)	✓
	Deklarative Validierung ist möglich	(K 3.4-3)	✗
	Fehlermeldungen können angezeigt werden	(K 3.4-4)	✓
Daten-Bindung	CommandBean ist Servlet- und Framework-unabh.	(K 3.5-1)	✓
	Typüberprüfungen und Typkonvertierungen	(K 3.5-2)	✓
	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	(K 3.4-3)	✓
	Falsche Daten werden im Formular angezeigt	(K 3.4-4)	✓
Internationalisierung	Länder-Information wird zur Verfügung gestellt	(K 3.6-1)	✓
	Beschriftungen sind vom Java-Code getrennt	(K 3.6-2)	✓
	Fehlermeldungen sind vom Java-Code getrennt	(K 3.6-3)	✓
Anwendungs-spezifisches	JDBC-Abstraktions-Mechanismus	(K 3.8-1)	✓
	Deklarative Transaktionsverwaltung ist möglich	(K 3.8-2)	✓
	Verwaltung der Abhängigkeiten	(K 3.8-3)	✓
	Wizard-Unterstützung	(K 3.8-4)	✓
	Datei-Upload-Unterstützung	(K 3.8-5)	✓
Doku-mentation	Referenz-Dokumentation	(K 3.7-1)	✗
	JavaDocs	(K 3.7-2)	✓
	Beispielanwendungen und Tutorials	(K 3.7-3)	✓
Ergebnis	Erfüllte Kriterien:	**25 von 27**	

Tabelle 2: Spring in Bezug auf die Evaluierungskriterien

6. Kapitel

Maverick

Ein weiteres aktionsgesteuertes Open-Source Java Web-Framework ist Maverick. Der Name entstand aus einem Wortspiel mit MVC. Es handelt sich bei diesem um ein minimalistisches Web-Framework, das nur die Basis-MVC-Funktionalitäten bietet. Es hat somit eine kleine Lernkurve und weist eine geringere Komplexität auf als die anderen in dieser Arbeit beschriebenen Frameworks. Im folgenden Abschnitt wird Maverick in der momentan aktuellen Version 2.2 beschrieben.

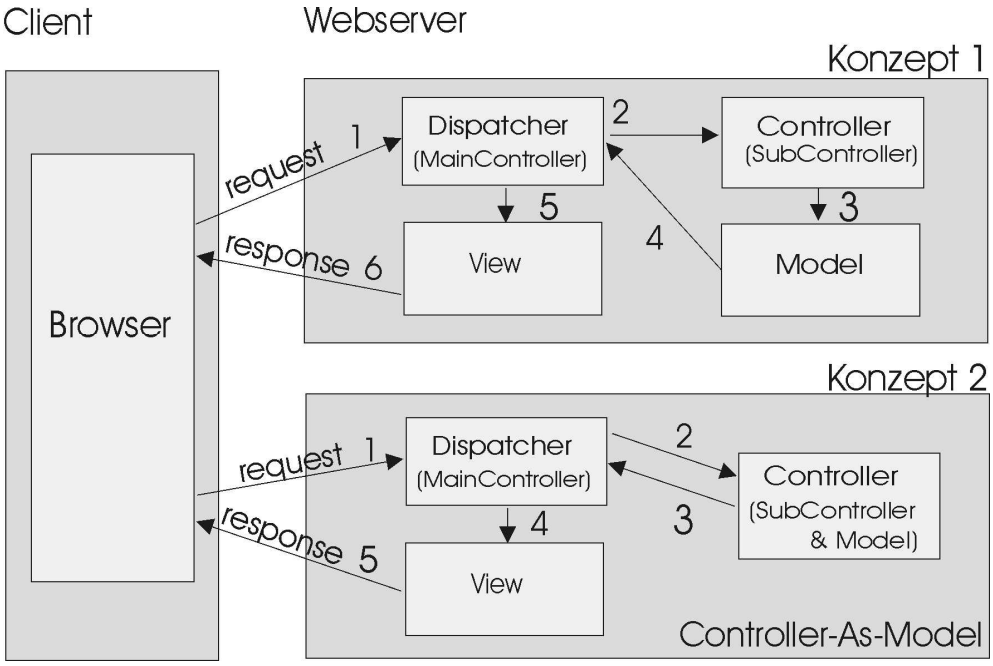

Abb. 6-1: Zwei mögliche Abläufe in Maverick

Der Main-Controller heißt in diesem Framework *Dispatcher*. Er behandelt die eingehenden Requests (1) und leitet sie an den entsprechenden Sub-Controller weiter (2). Bis zu diesem Punkt sind die zwei Möglichkeiten des Framework-Ablaufes gleich. Im ersten Ablaufkonzept befüllt der Sub-Controller das Model (3), das die Daten für die Anzeige zur Verfügung stellt und gibt die Kontrolle wieder an den Main-Controller zurück. Von diesem wird dann die richtige View ausgewählt (4), mit Hilfe der Model-Daten aufgebaut (5) und schließlich im Browser angezeigt (6). Der Unterschied im zweiten Ablaufkonzept liegt in (3), der Sub-Controller befüllt nicht das Model, sondern fungiert selbst als Model und versorgt sich deshalb selbst mit den notwendigen Daten.

6.1 Controller

Auch in Maverick werden die nicht generischen Controller-Aufgaben vom Main-Controller in Hilfs-Klassen, so genannte Sub-Controller auslagert. Deshalb wird die Rolle des Controllers in zwei Abschnitten – Main-Controller und Sub-Controller – beschrieben.

6.1.1 Main-Controller (Dispatcher)

Der erste Schritt um eine Maverick-Anwendung zu entwickeln ist, wie auch bei den anderen Frameworks, den Main-Controller, hier *Dispatcher* genannt, im Deployment-Descriptor (*web.xml*) der Web-Applikation zu deklarieren. Abb. 6-2 zeigt diese Deklaration:

```
<servlet>
  <servlet-name>maverick</servlet-name>
  <display-name>Maverick Dispatcher</display-name>
  <servlet-class>org.infohazard.maverick.Dispatcher
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>maverick</servlet-name>
  <url-pattern>maverick/*.m</url-pattern>
</servlet-mapping>
```

Abb. 6-2: Auszug aus web.xml

Im Abschnitt <*servlet*> wird die eigentliche Deklaration vorgenommen, es wird ein Name für den Main-Controller angegeben und die generische Framework-Klasse

spezifiziert. Anschließend, im <*servlet-mapping*>-Bereich wird durch das „url-pattern“ der Pfad beschrieben mit dem der Main-Controller erreicht werden kann.

Bei der Initialisierung des Servlets wird standardmäßig nach einer *maverick.xml*-Datei im WEB-INF-Verzeichnis gesucht. Die Konfigurations-Datei beinhaltet drei Hauptelemente: Modules, Views und Commands. Das <*modules*>-Element fungiert als Container für Plug-Ins für Erweiterungen. Im <*views*>-Element können globale Views definiert werden und im <*commands*>-Element wird die Zuordnung von Controllern und Views vorgenommen. Durch diese externe Zuordnung ist der Controller-Code an keine View-Technologie gebunden. Das ermöglicht ein einfaches Austauschen der View. Die Konfigurations-Datei von Maverick wird in Abb. 6-3 dargestellt:

```
<commands>
  <command name="showShedule">
    <controller class="eval.maverick.ViewSchedule"/>
    <view name="success" path="ScheduleView.jsp"/>
  </command>
  <command name="scheduleEntry">
    <controller class="eval.maverick.ScheduleEntry"/>
    <view name="success" path="ScheduleEntryView.jsp"/>
  </command>
  <command name="addToSchedule">
    <controller class="eval.maverick.AddToSchedule"/>
    <view name="error" path="scheduleEntry.do"/>
    <view name="success" path="showSchedule.do"/>
  </command>
</commands>
```

Abb. 6-3: Auszug aus maverick.xml

Die in Abb. 6-3 gezeigte Konfigurations-Datei der Beispiel-Applikation enthält nur das dritte der Hauptelemente, da die anderen in diesem Fall nicht notwendig sind. Es werden drei Sub-Controller deklariert (siehe 6.1.2). Der *showSchedule* Sub-Controller ist für die Anzeige des Terminplans verantwortlich, deshalb wird ihm die View „*ScheduleView.jsp*“ zugeordnet. Er besitzt nur eine *success*-View, da in jedem Fall diese Datei angezeigt werden soll. Der *scheduleEntry* Sub-Controller ermöglicht die Eingabe eines neuen Termins mittels der Formularseite „*ScheduleEntryView.jsp“*. Der *addToSchedule* Sub-Controller fügt den neuen Eintrag in der Datenbank den bereits bestehenden hinzu. Falls er dabei erfolgreich ist (*success)*, wird die Kontrolle wieder an den *showSchedule* Sub-Controller übergeben, falls nicht (*error*), muss die

Formularseite erneut angezeigt werden, wofür der *scheduleEntry* Sub-Controller zuständig ist.

Dadurch, dass zwei Controller notwendig sind, um die Formularabwicklung zu erledigen – es gibt in Maverick keine eigene Art Formular-Controller - wird das Kriterium K 3.1-2 nicht erfüllt.

6.1.2 Sub-Controller (Controller)

Ein Sub-Controller ist eine Hilfs-Klasse des Main-Controllers, die Web-Applikations-spezifische Aufgaben übernimmt. Im Folgenden wird statt Sub-Controller der Begriff Controller verwendet. In Maverick kann für die Controller-Implementierung zwischen verschiedenen Konzepten gewählt werden. Es kann einerseits zwischen Singleton und Nicht-Singleton-Controllern gewählt werden und andererseits, ob das „Model-As-Controller-Pattern“ verwendet wird oder nicht. Beide Konzepte werden in den nächsten Absätzen kurz erklärt. In jedem Fall ist jedoch der Controller eine JavaBean, womit das Kriterium 3.1-1 erfüllt wird.

Ein Singleton ist ein Objekt, das nur einmal in der ganzen Web-Applikation instanziert wird. Somit wird es von mehreren Threads verwendet. Damit es nicht zu unerwünschten Resultaten kommen kann, darf der Controller keine Attribute enthalten, die während der Applikation verändert werden. Model-Daten werden deshalb in einem anderen Objekt gespeichert. Dieses Objekt wird in Maverick *FormBean* genannt (siehe Abschnitt 6.2).

Verwendet ein Framework das „Model-As-Controller-Pattern“, so bedeutet das, dass das Model und der Controller in einer Komponente zusammengefasst werden. Der Controller übernimmt gleichzeitig die Rolle des Models. Nicht die Daten eines externen Objekts werden untersucht oder befüllt, sondern die Attribute des Controllers. Die Entscheidung für oder gegen dieses Entwurfsmuster würde bestimmen, ob das Kriterium 3.2-3 erfüllt wird oder nicht. Da aber die Möglichkeit besteht, das Model in einer eigenen Komponente zu realisieren, wird das Kriterium 3.2-3 in dieser Evaluierung als erfüllt betrachtet.

Nach der Erklärung der beiden zugrunde liegenden Konzepte, werden in den folgenden Absätzen die Möglichkeiten erläutert, wie diese Konzepte umgesetzt werden können. Die Menge der von Maverick zur Verfügung gestellten Sub-Controller kann in zwei Bereiche eingeteilt werden - in „Singleton“- und „Nicht-Singleton“-Controller.

Erstens gibt es die „Singleton“-Controller, die das Interface *ControllerSingleton* implementieren. Zur Vereinfachung der Implementierung kann die abstrakte Klasse *FormBeanUser* erweitert werden. Wie der Klassenname schon vermuten lässt, verwendet diese Controller-Art eine *FormBean*, ein externes Model-Objekt. Folgende Methode muss überschrieben werden, um die Controller-Funktionalität zu implementieren:

```
protected String perform(Object formBean,
              ControllerContext cctx) throws Exception
```

Abb. 6-4: Methode der FormBeanUser-Klasse

Der *perform()*–Methode aus Abb. 6-4 wird als erstes Argument ein *FormBean*-Objekt übergeben, das Model-Daten enthält (siehe Kapitel 6.2). Diese *FormBean* wird in dieser Methode an die Geschäftslogik übergeben, um entweder die vom Benutzer gelieferten Daten in die Datenbank zu speichern oder damit sie mit den zur späteren Anzeige notwendigen Daten befüllt werden kann. Das zweite Argument ist ein *ControllerContext*-Objekt. Diese JavaBean beinhaltet alle Web-Applikations-bezogenen Informationen wie z.B. den Request, den Response etc. Es wird dazu verwendet, jene Request-Daten zu gewinnen, die nicht automatisch gesetzt werden konnten (siehe Abschnitt 6.5). Schließlich gibt die Methode einen String zurück, der die anzuzeigende View bezeichnet.

Zweitens gibt es den „Nicht-Singleton“-Ansatz, dabei werden so genannte „Throw-Away“-Controller verwendet, die das Interface *Controller* implementieren. Es wird bei jedem Request eine neue Controller-Instanz angelegt. Von der zweiten Controller-Art stellt Maverick drei verschiedene Superklassen zur Verfügung: *ThrowawayBean2*, *ThrowawayFormBeanUser* und *ControllerWithParams*.

ThrowawayBean2 ist ein ganz einfacher Controller, der zugleich als Model funktioniert. Er implementiert das „Model-As-Controller-Pattern“. Das wird erreicht, indem der Controller nicht nur eine *perform()*- Methode zur Ausführung der Applikationslogik, sondern auch Zugriffs- und Änderungs-Methoden für die in ihm enthaltenen Model-Daten zur Verfügung stellt. Nach Ausführung seiner *perform()*-Methode, wird der gesamte Controller unter dem Schlüssel “model“ an den Request gebunden.

ControllerWithParams implementiert ebenfalls das „Model-As-Controller-Pattern“ und ist somit zugleich Model und Controller. Er kann jedoch mittels der *maverick.xml*-Datei parametrisiert werden. Das hat den Vorteil, dass gewisse Änderungen - wie z.B. die Änderung der View - nur in der XML-Datei vorgenommen

werden müssen und nicht im Java-Code. Kriterium 3.3-2 wird dadurch erfüllt. Die Applikation wird flexibler und besser konfigurierbar. Bei Verwendung dieser Klasse muss an Stelle der *perform()*-Methode die Methode *go()* implementiert werden.

ThrowawayFormBeanUser ist eine Mischform aus *ThrowawayBean2* und *FormBeanUser*. Er bietet, im Gegensatz zu den anderen beiden „Nicht-Singleton"-Controllern die Möglichkeit, ein separates Objekt als Model zu benutzen. Wird dieser Controller als Superklasse verwendet, wird entweder nur die *perform()*-Methode oder auch die Methode *makeFormBean()* überschrieben. Letztere gibt eine einfache Bean zurück, die die Rolle des Models übernehmen soll. Standardmäßig wird der gesamte Controller zurückgegeben. Abb. 6-5 zeigt den *AddToSchedule*-Controller der Beispiel-Applikation (vergleiche Abschnitt 2.3):

```
public class AddToShedule extends ControllerWithParams {
  private ScheduleItemValidator val;
  private Map errors;
  private ScheduleDB sdb;
  private ScheduleItem si;
  […] // siehe Kapitel 6.2
  public String go(ControllerContext ctx)throws Exception{
    errors = new HashMap();
    val.validateText(text, errors);
    val.validateDuration(duration, errors);
    val.validateStart(start, errors);
    val.validateEventType(eventType,errors);
    if (errors.isEmpty()) {
      int d= Integer.parseInt(duration);
      si= new ScheduleItem(text, d, start, eventType);
      sdb = new ScheduleDb();
      sdb.setDriver((String) params.get("driver"));
      sdb.setUrl((String) params.get("url"));
      sdb.setPw((String) params.get("pw"));
      sdb.setUser((String) params.get("user"));
      sdb.addRecord(si);
      return SUCCESS;
    }else{
      return ERROR;
    }
  }
}
```

Abb. 6-5: AddToSchedule als Controller

Der Abb. 6-5 kann entnommen werden, dass der Controller die Validierung der Felder selbst übernehmen muss (siehe Kapitel 6.4). Es ist eine eigene Klasse *ScheduleItemValidator* notwendig, für die das Framework keinerlei Vorgaben macht.

Mittels der *validate[Attributname]()*-Methoden dieser Klasse werden die Formularfelder auf Gültigkeit überprüft. Treten dabei Fehler auf, werden diese in ein Fehler-Objekt geschrieben, wofür vom Framework ebenfalls kein Interface zur Verfügung gestellt wird. Sind bei der Überprüfung keine Fehler aufgetreten, wird das *ScheduleItem*-Objekt angelegt und dem Datenbankzugriffs-Objekt übergeben. Dieses speichert es in die Datenbank. Auch das *ScheduleDb*-Objekt wird hier im Controller angelegt, die dazu notwendigen Parameter kommen aus der *maverick.xml*-Datei, da der Controller die Klasse *ControllerWithParam* erweitert. In diesem Fall kann nicht mehr von einer sauberen Schichtentrennung gesprochen werden, da der Controller keine datenbankbezogene Informationen enthalten sollte. Kriterium K 3.1-3 wird dadurch nicht erfüllt.

6.2 Model (Controller oder JavaBean)

Wie im vorigen Abschnitt schon erläutert wurde, gibt es zwei Möglichkeiten für das Model, je nachdem ob das „Model-As-Controller-Pattern“ angewandt wird oder nicht. Im Folgenden werden diese beiden Optionen näher erläutert. Die Controller, die das „Model-As-Controller-Pattern“ implementieren, enthalten neben der Methode, die den Controller-Code ausführt, auch noch Zugriffs- und Änderungsmethoden für die in ihnen enthaltenen Model-Daten. Abb. 6-6 zeigt den Teil des *AddToSchedule*-Controller der Beispiel-Applikation, die für die Model-Rolle relevant ist:

```
public AddToSchedule extends ControllerWithParams{
   […] //nicht-Model-Attribute → siehe Abb. 6-5
   private Map errors;
   private String start;
   private String duration;
   private String text;
   private String eventType;
   public void setStart(String s) {start=s;}
   public void setDuration(String d) {duration=d;}
   public void setText(String t) {text= t;}
   public void setEventType(String et) {eventType= et;}
   public String getStart() {return start;}
   public String getDuration() {return duration;}
   public String getText() {return text;}
   public String getEventType() {return eventType;}
   […] //go() → siehe Abb. 6-5
}
```

Abb. 6-6: AddToSchedule als Model

Wie in Abb. 6-6 zu sehen ist, enthält der *AddToSchedule*-Controller die Attribute *start*, *duration*, *text* und *eventType* und die entsprechenden Zugriffs- und Änderungs-Methoden. Außerdem enthält er ein *errors*-Objekt, das die Fehler beinhaltet, die gegebenenfalls angezeigt werden sollen. Der Controller enthält alle notwendigen Daten, die für die View zur Anzeige von Bedeutung sind. Der Controller macht diese Attribute der View zugänglich, indem er nach Ausführung der *perform()*- oder *go()*-Methode sich selbst mittels geeigneter Änderungsmethode an den *ControllerContext* übergibt. Die View kann dann die Zugriffsmethode des *ViewContext*-Objekts *getModel()* aufrufen, um die Model-Daten zur Verfügung zu haben.

Dadurch, dass der Controller zusätzlich auch die Rolle des Model übernimmt, wird nicht nur Kriterium 3.2-3 sondern auch 3.2-1 nicht erfüllt, da der Controller und somit auch das Model vom Framework und der Servlet-API abhängig sind. Es können jedoch beliebig viele JavaBeans der Geschäftslogik als Attribut im Controller vorkommen, was die Erfüllung des Kriteriums 3.2-2 bedeutet.

Die anderen Controller, bei denen auf eine saubere Rollen- und Kompetenzaufteilung wertgelegt wird, enthalten die Model-Daten nicht selbst, sondern erzeugen ein Objekt, das diese Informationen kapselt. Sie erfüllen damit Kriterium 3.2-3. Dieses Objekt wird *FormBean* genannt, muss aber nicht, wie in Struts, von einer Framework-Klasse abgeleitet werden, sondern kann Framework und Servlet-API-unabhängig implementiert werden. Das bedeutet, auch dem Kriterium 3.2-1 wird dieser Model-Ansatz gerecht. Abb. 6-7 zeigt als Beispiel die FormBean der Beispiel-Applikation:

```
public class ScheduleItem {
   private String start;
   private int duration;
   private String text;
   private String eventType;

   public ScheduleItem() {}
   public String getStart() {return start;}
   public void setStart(String start) {this.start = start;}
   […]
}
```

Abb. 6-7: ScheduleItem.java

In Abb. 6-7 ist dargestellt, dass als FormBean eine JavaBean der Geschäftslogik verwendet werden kann. Sind die Model-Daten jedoch auf verschiedene JavaBeans aufgeteilt, müsste ein zusätzliches Container-Objekt erstellt werden, das all diese JavaBeans enthält, wodurch das Kriterium 3.2-2 nicht erfüllt wäre. Es kann jedoch

eine Java-HashMap verwendet werden, die der Forderung des Kriteriums 3.2-2 gerecht wird.

Damit die Model-Daten auch automatisch gesetzt werden können, (siehe Kapitel 6.5 Daten-Bindung) und für die View zugänglich sind, muss die Methode *makeFormBean()* überschrieben werden, die dem Framework mitteilt, welches Objekt als Model verwendet werden soll. Abb. 6-8 zeigt diese Methode im *AddToSchedule*-Controller:

```
protected abstract Object makeFormBean
                                (ControllerContext ctx){
  return new ScheduleItem();
}
```

Abb. 6-8: Auszug aus AddToSchedule.java

In der in Abb. 6-8 dargestellten Methode wird ein *ScheduleItem*-Objekt zurückgegeben, das dadurch als Model-Objekt behandelt wird und nach der Ausführung der *go()*-Methode durch die Methode der *ControllerContext*-Klasse *setModel()* zugänglich gemacht wird. Durch den möglichen Zugriff auf das *ControllerContext*-Objekt, das der Methode als Argument übergeben wird, könnte auch ein Objekt als Model verwendet werden, das bereits in der Session existiert.

6.3 View

Mit Maverick ist es möglich JSP [Sun03a], XSLT [W3C99] oder Velocity [Velo04] als View-Technologien zu verwenden. Kriterium 3.2-1 wird dadurch erfüllt. Jede benannte View muss das Interface *View* implementieren, das auf eine View-Technologie-unabhängige und konsequente Art Model-Daten an jede beliebige View übergibt. Das Interface ist sehr einfach und beinhaltet nur eine Methode:

```
public void go(ViewContext vctx) throws Exception
```

Abb. 6-9: Einzige Methode des View-Inferface

So können View-Komponenten für andere View-Technologien programmiert werden. Im *ViewContext*-Objekt ist, genau wie im *ControllerContext*, das Model via der *getModel()*-Methode zu erreichen und so werden Request als auch Response durch dieses Objekt für die View zugänglich gemacht. Die Standard-Implementierung für JSP-Views bindet das Model über ein Attribut an den Request, bevor zur JSP-Seite weitergeleitet wird.

Damit XSLT verwendet werden kann, bietet Maverick ein Modul zur transparenten Transformation in DOM-Objekten [W3C00] an, mit dem JavaBeans effektiv zu XML-Dateien konvertiert werden. Außerdem wird noch eine konfigurierbare Transformations-Pipeline geboten, die für XSLT brauchbar ist, jedoch für die anderen View-Technologien uninteressant ist.

6.4 Formular-Validierung

Maverick stellt keinerlei Unterstützung für die Validierung zur Verfügung. Dafür ist alleine der Controller verantwortlich. Das bedeutet, dass keines der geforderten Kriterien erfüllt wird. Der Controller muss auch selbst die Typüberprüfungen und Konvertierungen durchführen, um ServletExceptions zu entgehen und sich um die eventuell auftretenden Ausnahmen kümmern (siehe Kapitel 6.5).

Es werden vom Framework auch keine Custom-Tags geliefert, die es erleichtern würden, von der View auf den Fehlercode zuzugreifen. Ist der Controller jedoch zugleich auch Model, hat die View einfachen Zugriff auf die Fehler, weil diese im Controller gespeichert sind.

6.5 Daten-Bindung

Unter Daten-Bindung wird der Vorgang des automatischen Befüllens von JavaBean-Properties mit Request-Parameter-Werten unter Verwendung von Reflection verstanden. Die zu befüllende JavaBean wird CommandBean genannt. In Maverick funktioniert Daten-Bindung sowohl bei den als „Model-As-Controller“-Pattern implementierten Controllern als auch bei den anderen. Mit Hilfe des Apache *BeanUtil*-Packages werden die Attribute der CommandBean durch die entsprechenden Änderungsmethoden automatisch gesetzt.

Als CommandBean wird vom Framework jenes Objekt erkannt, das im *ControllerContext* unter dem Property *model* erreichbar ist. In dem einen Fall wird die Rolle der CommandBean vom Controller übernommen, nämlich dann, wenn das „Controller-As-Model“-Pattern verwendet wird. Somit ist die CommandBean vom Framework und von der Servlet-API abhängig und kann nicht direkt an die Geschäftslogik weitergegeben werden. Kriterium K 3.5-1 ist in diesem Fall nicht erfüllt. Da jedoch im anderen Fall das Model die Rolle der CommandBean übernimmt und es somit möglich ist, dieses Kriterium zu erfüllen, genügt Maverick diesem Kriterium K 3.5-1.

Was am Beispiel der Beispiel-Applikation in Abb. 6-6 und Abb. 6-7 auffällt ist, dass die Model-Attribute alle vom Typ String sind. Das ist deshalb so, um der Gefahr einer ServletException zu entgehen. Da das Apache Bean-Manipulations-Konzept verwendet wird, könnte bei der Konvertierung von dem String Request-Parameter *duration* in das int-Feld *duration* ein in einer ServletException resultierender Fehler auftreten. Das bedeutet die Nicht-Erfüllung der Kriterien K 3.5-2 und K 3.5-3. Der Controller kann nur die Properties validieren, die erfolgreich gesetzt worden sind. Das Framework stellt außerdem keinen Mechanismus zur Verfügung, um dem/der BenutzerIn den zuvor falsch eingegebenen Wert erneut anzuzeigen. Maverick erfüllt deshalb das Kriterium 3.5-4 nicht.

6.6 Internationalisierung

Maverick unterstützt Internationalisierung insofern, dass für jede Sprache, in der die Webseiten angezeigt werden sollen, eine eigene HTML Seite erzeugt werden muss. In der Konfigurations-Datei von Maverick kann mit Hilfe des Properties *mode* angegeben werden, welche Seite zu welcher Sprache gehört. Gemäß der Browser-Orts-Information wählt Maverick dann entweder die Seite mit der gewünschten Sprache oder, wenn diese nicht vorhanden ist, die Standardseite aus. Dadurch erfüllt das Framework die Kriterien K 3.6-1 und K 3.6-2.

Das Problem dabei ist, dass viele Seiten gewartet werden müssen. Die Möglichkeit Fehlermeldungen zu internationalisieren wird vom Framework nicht unterstützt, was natürlich ein großes Manko darstellt und das Kriterium K 3.6-3 nicht erfüllt.

6.7 Anwendungsspezifisches

Da Maverick sich auf die wesentlichen Funktionen eines Web-Frameworks beschränkt, beinhaltet es keine Unterstützung für die Datenbankanbindung. Dasselbe gilt auch für Transaktionsunterstützung, die Datei-Upload-Unterstützung, externe Verwaltung der Objekte der Geschäftslogik und für Wizard-Controller. Maverick erfüllt somit keines der anwendungsspezifischen Kriterien.

6.8 Dokumentation

Obwohl Maverick ein sehr leichtes und minimalistisches Framework ist, ist die Referenz-Dokumentation enttäuschend, da sie sehr viele Fragen unbeantwortet lässt. Maverick wird Kriterium K 3.8-1 somit nicht gerecht. Auch die JavaDocs sind wenig kommentiert und wenn, dann nicht ausreichend detailliert. Das bedeutet dass Kriterium K 3.8-2 ebenfalls nicht erfüllt wird. Es wird eine vollständige Anwendung als Beispiel-Applikation zur Verfügung gestellt, die sowohl in dem „Singleton"- als auch im „Nicht-Singleton"-Ansatz implementiert ist. Kriterium 3.8-3 wird dadurch erfüllt.

Dadurch, dass die Klassenanzahl jedoch relativ klein ist, ist der Maverick-Code doch überschaubar. Durch das klare Konzept und die Verwendung von Interfaces ist der Umgang mit diesem Framework trotz der nicht hervorragenden Dokumentation leicht zu erlernen.

6.9 Zusammenfassung

Folgende Tabelle soll zum Abschluss der Beschreibung des Maverick-Frameworks eine zusammenfassende Übersicht über die Eigenschaften von Maverick in Bezug auf die Evaluierungskriterien geben:

Controller	Sub-Controller sind JavaBeans	(K 3.1-1)	✓
	Superklassen für zwei verschiedene Kontrollflüsse	(K 3.1-2)	×
	Controller enthalten keine Datenbank-Informationen	(K 3.1-3)	×
Model	Model ist Servlet- und Framework-unabhängig	(K 3.2-1)	✓
	Model kann mehrere JavaBeans enthalten	(K 3.2-2)	✓
	Model ist durch eigene Komponente realisiert	(K 3.2-3)	✓
View	Verschiedene View-Technologien werden unterstützt	(K 3.3-1)	✓
	View-Austausch ist ohne Controller-Änderung möglich	(K 3.3-2)	✓
Formular-Validierung	Validatoren sind Web-unabhängige JavaBeans	(K 3.4-1)	×
	Es können eigene Validatoren implementiert werden	(K 3.4-2)	×
	Deklarative Validierung ist möglich	(K 3.4-3)	×
	Fehlermeldungen können angezeigt werden	(K 3.4-4)	×
Daten-Bindung	CommandBean ist Servlet- und Framework-unabh.	(K 3.5-1)	✓
	Typüberprüfungen und Typkonvertierungen	(K 3.5-2)	×
	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	(K 3.4-3)	×
	Falsche Daten werden im Formular angezeigt	(K 3.4-4)	×
Internationalisierung	Länder-Information wird zur Verfügung gestellt	(K 3.6-1)	✓
	Beschriftungen sind vom Java-Code getrennt	(K 3.6-2)	✓
	Fehlermeldungen sind vom Java-Code getrennt	(K 3.6-3)	×
Anwendungs-spezifisches	JDBC-Abstraktions-Mechanismus	(K 3.8-1)	×
	Deklarative Transaktionsverwaltung ist möglich	(K 3.8-2)	×
	Verwaltung der Abhängigkeiten	(K 3.8-3)	×
	Wizard-Unterstützung	(K 3.8-4)	×
	Datei-Upload-Unterstützung	(K 3.8-5)	×
Doku-mentation	Referenz-Dokumentation	(K 3.7-1)	×
	JavaDocs	(K 3.7-2)	×
	Beispielanwendungen und Tutorials	(K 3.7-3)	✓
Ergebnis	Erfüllte Kriterien:	**10 von 27**	

Tabelle 3: Maverick in Bezug auf die Evaluierungskriterien

7. Kapitel

WebWork2

Das aktionsgesteuerte MVC-Framework der OpenSymphonie-Gruppe ist momentan in der Version 2.1 verfügbar [Webw04]. Der Sprung von der ersten auf die zweite Version des Frameworks hat große Änderungen mit sich gebracht. WebWork2 besteht jetzt aus zwei Haupt-Komponenten. Aus dem früheren WebWork hat sich der MVC-Kern des Frameworks abgespalten und wurde zu einem eigenständigen Projekt. Diese neue Komponente heißt XWork. Abb. 7-1 zeigt den aktionsgesteuerten Kontrollfluss, der in zwei Varianten möglich ist:

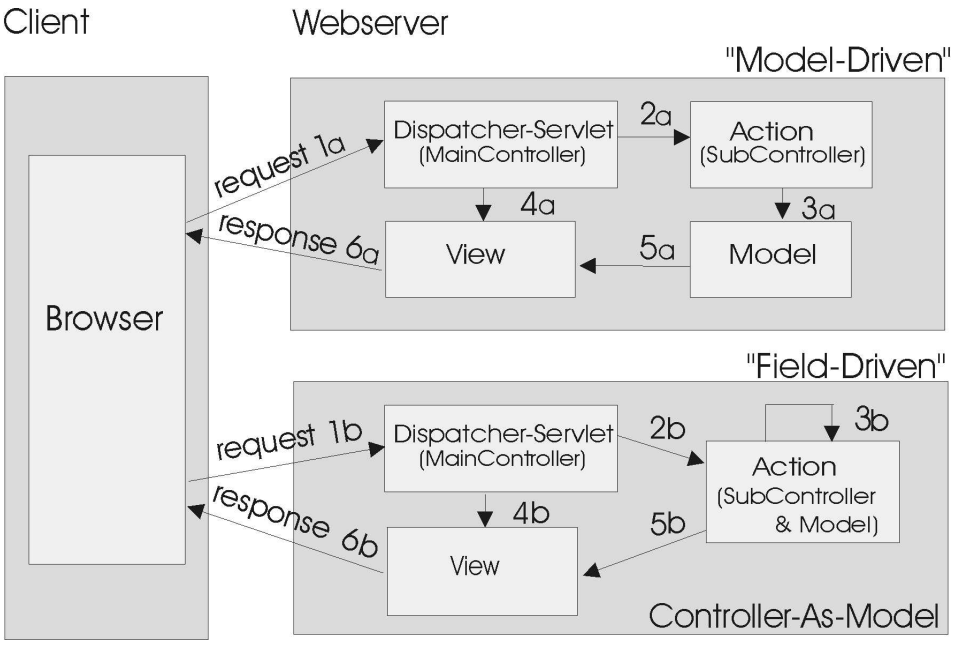

Abb. 7-1: MVC-Komponenten von WebWork2

Den Einstiegspunkt des Frameworks bildet der Main-Controller, der in WebWork2 als *Dispatcher-Servlet* bezeichnet wird. Dieser leitet die eingehenden Requests (1ab) an die entsprechenden *Actions* weiter (2ab), die nicht generische Controller-Aufgaben übernehmen. Eine *Action* ist somit ein Sub-Controller. Sie befüllt entweder eine JavaBean (3a), die Model-Daten zur Verfügung stellt, oder sich selbst (3b) und gibt die Kontrolle wieder an das *ActionServlet* zurück. Von diesem wird dann die anzuzeigende View ausgewählt (4ab). Mit Hilfe der Model-Daten, die entweder in einem externen Model-Objekt („Model-Driven“ 5a) oder im Sub-Controller („Field-Driven“ 5b) vorliegen, wird die View im Browser angezeigt (6ab).

7.1 Controller

Wie in Abb. 7-1 schon zu sehen war, werden die eingehenden Requests nicht nur von einem Controller verarbeitet, sondern von einem Main-Controller an so genannte Sub-Controller weitergereicht. Deshalb wird die Rolle des Controllers in dieser Arbeit in zwei Abschnitten – Main-Controller und Sub-Controller – beschrieben.

7.1.1 Main-Controller (Dispatcher-Servlet)

Die generische, vom Framework gelieferte Klasse *ServletDispatcher* stellt den Main-Controller dar. Sie wird, wie jedes Servlet, im Deployment-Descriptor der Web-Applikation (*web.xml*) deklariert. Abb. 7-2 zeigt diese Deklaration:

```
<servlet>
   <servlet-name>webwork</servlet-name>
   <servlet-class> webwork.dispatcher.ServletDispatcher
   </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>webwork</servlet-name>
   <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

Abb. 7-2: Auszug aus web.xml

Innerhalb des <*servlet*>-Elements werden für das Servlet ein Name und die implementierende Framework-Klasse angegeben. Durch das <*servlet-mapping*> wird angegeben, durch welche Pfade das Servlet erreicht werden kann. Dazu wird im Abschnitt <*url-pattern*> beschrieben, nach welchem Muster eine URL aufgebaut sein muss, damit es von diesem Servlet verarbeitet wird. Beim Start liest das Dispatcher-Servlet XML- und andere Konfigurations-Dateien. Die Datei *xwork.xml* ist die erste

davon, da in ihr alle anderen Konfigurations-Informationen enthalten sind. Abb. 7-3 zeigt einen Ausschnitt von *xwork.xml* der Beispiel-Applikation (vergleiche Abschnitt 2.3):

```
<include file="webwork-default.xml" />
<package name="default" extends="webwork-default">
  <default-interceptor-ref name="defaultStack" />
  <action name="viewSchedule"
    class="vergleich.webwork.ViewSchedule">
    <result name="success" type="dispatcher">
      ScheduleView.jsp
    </result>
    <interceptor-ref name=”dbP”/>
  </action>
  <action name="scheduleEntry"
    class="vergleich.webwork.ScheduleEntry">
    <result name="success" type="dispatcher">
      ScheduleEntryView.jsp
    </result>
  </action>
  <action name="addSchedule"
    class="vergleich.webwork.AddScheduleEntry">
    <result name="success" type="chain">viewSchedule
    </result>
    <result name="error" type="chain">scheduleEntry
    </result>
    <interceptor-ref name=”dbP”/>
  </action>
</package>
<interceptors>
  <interceptor name=“dbP“
    class=“vergleich.webwork.DbParamInterceptor/>
</interceptors>
```

Abb. 7-3: Auszug aus xwork.xml

Wie in Abb. 7-3 ersichtlich ist, wird zuerst die *webwork-default.xml* Datei inkludiert, denn darin sind alle Standard-Konfigurations-Einstellungen enthalten. Danach wird in dem Element <*package*> das Standard-Package *webwork-default* definiert, in dem alle Sub-Controller-Komponenten registriert werden. Darin wird der *defaultStack* angegeben, der intern von WebWork2 für die Interceptoren (siehe Kapitel 7.1.2) verwendet wird. Die verschiedenen Sub-Controller werden durch ein <*action*>-Element definiert. Das darin liegende <*result*>-Element sagt aus, welche View (property *name*) und welche Art von View (property *type*) zur Anzeige verwendet werden soll. Die Ausprägung *dispatcher* bedeutet, dass direkt zu einer Web-Seite weitergeleitet wird. Bei der Ausprägung *chain* wird die Kontrolle an den definierten

Sub-Controller übergeben. Durch die Konfiguration der Ergebnis-Views in der XML-Datei, kann ganz einfach die View ausgetauscht werden, ohne dass eine Änderung im Controller-Code notwendig ist. Kriterium K 3.2-2 wird dadurch erfüllt. Im Abschnitt <*interceptors*> werden die zur Applikation gehörigen Interceptoren definiert, die von den Sub-Controllern durch <*interceptor-ref*> referenziert werden können.

Für die Beispiel-Applikation müssen drei Sub-Controller deklariert werden, denn die Aufgaben des Anzeigens des Formulars und dessen Verarbeitung müssen in zwei verschiedenen Klassen erledigt werden (*ScheduleEntry* und *AddScheduleEntry*). Es wird vom Framework kein Sub-Controller zur Verfügung gestellt, der den gesamten Formular-Kontrollfluss übernehmen kann, somit wird das Kriterium K 3.1-2 nicht erfüllt. Die Sub-Controller *ViewSchedule* und *AddScheduleEntry* benötigen die Dienste des Interceptors *DBParamInterceptor* (siehe Abschnitt 7.1.2), deshalb wird dieser von ihnen referenziert.

7.1.2 Sub-Controller (Actions und Interceptoren)

In der Entwicklung von WebWork2 ist die Web-Unabhängigkeit der Komponenten in den Vordergrund gestellt worden. Das gesamte Teil-Projekt XWork ist vollständig von der Servlet-API unabhängig. Um Web- oder auch Datenbank-spezifischen Code aus dem Sub-Controller auszulagern wird das Konzept der Interceptoren verwendet.

Die nicht generischen Controller-Aufgaben werden deshalb in diesem Framework von zwei verschiedenen Komponenten übernommen. Web-unabhängige Aufgaben werden in so genannten *Actions* realisiert, während web-spezifischer Code in separate Klassen, die *Interceptoren* genannt werden, ausgelagert wird.

Eine *Action* ist eine JavaBean, wodurch das Kriterium K 3.1-1 erfüllt ist. In WebWork2 werden Actions nicht als „Singletons“ (vergleiche Kapitel 6.2) realisiert, das heißt eine *Action* kann nicht wieder verwendet werden. Der Sub-Controller muss das *Action*-Interface implementieren und die Methode *execute()* zur Verfügung stellen. Diese liefert als Rückgabewert einen String-Parameter, der die anzuzeigende View repräsentiert. Dieser Parameter muss sich mit dem Ergebnistyp der *Action*, wie er in dem <*result*>-Element der *xwork.xml* Datei konfiguriert wurde, decken. Das Dispatcher-Servlet kann somit die richtige View auswählen.

Zur Erleichterung der *Action*-Implementierung stellt WebWork2 eine Super-Klasse *ActionSupport* zur Verfügung. Diese gibt Unterstützung bei der Umsetzung von

Internationalisierung, Fehlerbehandlung und Logging. In Abb. 7-4 wird die Implementierung einer *Action* der Beispiel-Applikation gezeigt:

```
public class AddScheduleEntry  extends ActionSupport
                               implements ModelDriven {
   private ScheduleDb scheduleDb;
   private SchedlueItem si;
   public void setScheduleDb(ScheduleDb sdb) {
      scheduleDb= sdb;
   }
   protected String execute() throws Exception {
      scheduleDb.addRecord(si);
      return SUCCESS;
   }
   public Object getObject() {
      si= new ScheduleItem();
      retrun si;
   }
}
```

Abb. 7-4: Auszug aus AddScheduleEntry.java

Die Action *AddScheduleEntry* ist dafür verantwortlich, die eingegebenen Daten in die Datenbank zu speichern. In ihrer *execute()*-Methode wird dazu die *addRecord()* – Methode des *ScheduleDb*-Objekts aufgerufen. Die Erzeugung des Datenbank-Services findet nicht im Sub-Controller statt, sondern wird vom *DbParamInterceptor* (siehe Abb. 7-6) erledigt. Somit ist die *Action* frei von Datenbank-spezifischen Informationen und erfüllt dadurch das Kriterium K 3.1-3. Durch die Methode *getObject()* wird das *ScheduleItem* an das Formular gebunden (siehe Abschnitt 7.5), so werden seine Attribute vom Framework mit den Werten der Formularfelder befüllt. Durch den Rückgabewert *SUCCESS* wird im Browser die Seite angezeigt, die in der *xwork.xml*-Datei diesem Ergebniswert zugeordnet wurde (vergleiche Abb. 7-3). Die oben verwendete *ScheduleItem*-Klasse ist in Abb. 7-5 dargestellt:

```
public class ScheduleItem {
   private String start;
   private int duration;
   private String text;
   private String eventType;

   public ScheduleItem() {}
   public String getStart() {return start;}
   public void setStart(String start) {this.start = start;}
   […]
}
```

Abb. 7-5: ScheduleItem.java

Wie in Abb. 7-5 ersichtlich ist, besteht die *ScheduleItem*-Klasse lediglich aus vier Attributen, ihren Zugriffs- und Änderungsmethoden und einem Konstruktor.

Ein Interceptor ist eine Klasse, die das *Interceptor*-Interface implementiert oder die Klasse *AroundInterceptor* erweitert. Der Interceptor hat Zugriff zur *Action* und zum *ActionContext*. Je nachdem, ob vor der *execute()*-Methode der *Action* oder danach Interceptor-Funktionen notwendig sind, werden die Methoden *intercept()*, *before()* und/oder *after()* überschrieben. Interceptoren werden in der *xwork.xml*-Datei deklariert, und jede *Action*, die einen Interceptor verwenden will, benötigt ein <*interceptor-ref*>-Element in der Action-Deklaration. (vergleiche Abb. 7-3) Abb. 7-6 zeigt den oben schon erwähnten Interceptor der Beispiel-Applikation, der das Einlesen der Datenbank-Parameter und die Erzeugung des Datenbank-Service-Objekts *ScheduleDb* übernimmt:

```
public class DbParamInterceptor extends AroundInterceptor{

  private String url;
  private String driver;
  private String pw;
  private String user;
  protected void before(ActionInvocation inv)
                                    throws Exception{
    ServletContext c= ActionContext.getContext()
                                   .getServletContext();
    driver= c.getInitParameter("dirver");
    pw= c.getInitParameter("pw");
    url= c.getInitParameter("url");
    user= c.getInitParameter("user");
    inv.getAction().setScheduleDb(
                  new ScheduleDb(url,driver,user,pw));
  }
  […] //intercept(), after()
}
```

Abb. 7-6: Auszug aus DbParamInterceptor.java

Wie in Abb. 7-6 ersichtlich ist, wird im Interceptor der Beispiel-Applikation die Methode *before()* überschrieben, da das Datenbank-Service-Objekt schon funktionsfähig sein muss, wenn die *execute()* Methode der *Action* aufgerufen wird. Die Klasse *ActionContext* liefert dem Interceptor den Zugang zu den Web-spezifischen Klassen, wie z.B. dem *ServletContext*, durch den die in der *web.xml* definierten *init*-Parameter erreicht werden können. Mittels der Klasse *ActionInvocation* hat der Interceptor auch Zugriff zur *Action* und kann somit das *ScheduleDB*-Objekt mit der zur Verfügung gestellten Änderungs-Methode setzen.

7.2 Model (Action oder JavaBean)

In WebWork2 gibt es zwei Arten von Actions, Field-Driven und Model-Driven Actions. Erstere verwenden das „Model-As-Controller-Pattern“ (vergleiche Abschnitt 6.2), in dem der Controller auch die Rolle des Models wahr nimmt, dadurch dass er neben seiner *execute()*- Methode auch noch Properties mit entsprechenden Zugriffs- und Änderungs-Methoden zur Verfügung stellt. Diese Daten sind in WebWork2 nicht auf Standard-Datentypen beschränkt, sondern können auch Listen oder eigene Objekte sein, die wiederum Attribute besitzen. So kann das Model mehrere JavaBeans der Geschäftslogik beinhalten, Kriterium K 3.2-2 ist dadurch erfüllt. Es ist jedoch keine eigene Komponente und hängt vom Framework ab, was die Verletzung der Kriterien K 3.2-1 und K 3.2-3 bedeutet.

Model-Driven Actions unterstützen die Code-Wiederverwendung durch Trennung des Aufgabenbereiches von Controller und Model. Die Action implementiert das *ModelDriven*-Interface und stellt die Methode *getObject()* zur Verfügung. Diese gibt das Objekt zurück, das als Model behandelt werden soll (Kriterium K 3.2-3). Da jedes beliebige Objekt zurückgegeben werden kann, ist das Model sowohl vom Framework als auch von der Servlet-API unabhängig (Kriterium K 3.2-1) und kann auch als Map realisiert werden, damit mehrere JavaBeans im Model enthalten sein können, ohne dass ein Wrapper-Objekt implementiert werden muss (Kriterium K 3.2-2).

Die Entscheidung für oder gegen das „Model-As-Controller-Pattern“ bestimmt, ob alle geforderten Kriterien erfüllt werden oder nur eines. Da es jedoch möglich ist allen Anforderungen gerecht zu werden, werden für diese Evaluierung die Kriterien K 3.2-1 bis K 3.2-3 als erfüllt betrachtet.

7.3 View

WebWork2 unterstützt nicht nur JSP [Sun03a] als View-Technologie. Es ist möglich User-Interface-Template-Engines, wie Velocity [Velo04] und FreeMaker [Frem04], in WebWork2 zu integrieren. Das Kriterium K 3.3-1 wird dadurch erfüllt.

Der häufigste Weg den Frameworks gehen um Model-Daten einer View zugänglich zu machen, ist der einer Tag Library. WebWork2 stellt eine Reihe von Custom-Tags zur Verfügung. Diese sind in zwei Kategorien einzuordnen, zum einen gibt es User-Interface-Tags und zum anderen Non-User-Interface-Tags.

Non-User-Interface-Tags sind Tags, die allgemeine Funktionen wie den Zugriff auf Beans, Properties, Actions oder Bedingungen und Iterationen übernehmen. In Abb. 7-7 ist ein Beispiel für einen solchen Tag dargestellt:

```
<ww:property value="duration"/>
```

Abb. 7-7: Beispiel für ein Non-UI-Tag

Mit dem in Abb. 7-7 abgebildeten Tag ist es möglich ein Model-Attribut anzusprechen. In dem Beispiel wird auf das Attribut *duration* des *ScheduleItem*-Objekts (vergleiche Abb. 7-5) zugegriffen.

Für jedes HTML-Formular-Tag gibt es ein WebWork2 User-Interface-Tag, das an Stelle der Standard-HTML-Tags verwendet werden kann. Abb. 7-8 zeigt ein Beispiel für die Ersetzung eines Standard-Tags durch ein WebWork2-UI-Tag:

```
<tr>
  <td>Duration:</td>
  <td><input type="text" name="duration"></td>
</tr>

wird ersetzt durch:

<ww:textfield label="Duration" name="duration"/>
```

Abb. 7-8: Beispiel für ein UI-Tag

In dem Beispiel aus Abb. 7-8 wird das Standard-HTML „Textfeld"-Tag durch ein Custom-Tag von WebWork2 ersetzt. Das hat den Vorteil, dass erstens die Bezeichnung des Eingabefeldes gleichzeitig in dem Tag integriert ist (Property *label*). Das erleichtert die Internationalisierung, und trägt zu einer einheitlichen Benennung der Felder bei. Zweitens wird dadurch überprüft, ob der eingegebene Wert den richtigen Typ hat, um dem Property der JavaBean, die an das Formular gebunden ist (siehe Kapitel 7.5), zugewiesen werden zu können.

Die Sprache, die WebWork2 für diese Tags verwendet nennt sich *OGNL* (Object Graphic Navigation Language), die mit dem *OGNL ValueStack* zusammenarbeitet. Der *ValueStack* ist ein interessanter Ansatz, durch den WebWork2 und dessen Tag Library sich von allen anderen aktionsgesteuerten Frameworks unterscheidet (das ereignisgesteuerte Framework Tapestry verwendet ebenfalls *OGNL*). Der *ValueStack* ist ein Stack, der Request-Parameter-Werte nicht nur der View, sondern auch dem Controller zugänglich macht. Die Parameter-Werte werden mittels *push()*-Methode

auf den Stack geladen und können mit *pop()* wieder herunter genommen werden. Dieses „Stack“-Konzept wird von WebWork2 intern verwendet.

7.4 Formular-Validierung

WebWork2 bietet zwei verschiedene Arten der Validierung an. Erstens, Validierung durch die Methode *validate()* der Action, die entweder selbst Validierungscode enthält oder eine externe Validierungs-Komponente aufruft und eine Map zurückgibt, die alle aufgetretenen Fehler beinhaltet. Die zweite Art, Daten auf ihre Richtigkeit zu überprüfen, bietet das XWork Validation Framework. Dieses wird vor allem für die User-Interface-Validierung eingesetzt, das heißt für syntaktische Überprüfungen, Anwesenheit eines Wertes, Typkorrektheit und Ähnliches. Das Validierungs-Framework macht es möglich, Validierung in einer XML-Datei (<*NameDerAction*>-*validation.xml*) deklarativ zu erledigen, wodurch das Kriterium K 3.4-3 erfüllt wird.

Die verwendeten Validatoren sind Java-Klassen, die weder vom Framework noch von der Servlet-API abhängig sind. Sie implementieren das *Validator*-Interface, was die Erfüllung des Kriterium K 3.4-1 impliziert. WebWork2 stellt Validatoren für Standard-Typen (wie z.B. Integer) und häufig vorkommende Validierungsregeln (wie z.B. Pflichtfeld) zur Verfügung. Alle Validatoren sind in der *validator.xml*-Datei registriert. Wird ein Validator benötigt, der nicht im Framework integriert ist, kann das *FieldValidator*-Interface implementiert werden und der neue Validator in die *validator.xml*-Datei eingetragen werden. So können Objekte beliebiger Klassen validiert werden, wodurch das Kriterium K 3.4-2 erfüllt wird. Die Validatoren können dadurch auch in anderen Zusammenhängen wieder verwendet werden. Abb. 7-9 zeigt die deklarative Validierung an Hand der *AddToSchedule*-Action der Beispiel-Applikation:

```
<validators>
  <field name="duration">
    <field-validator type="requiredint">
      <param name="min">0</param>
      <param name="max">31</param>
      <message>
        duration must be between ${min} and ${max}!
      </message>
    </field-validator>
  </field>[…]
<validators>
```

Abb. 7-9: Auszug aus AddToSchedule-validation.xml

Wie in Abb. 7-9 zu sehen ist, können im Abschnitt <*validators*> die zu validierenden Felder angeführt werden. Das Beispiel stellt dar, wie die Validierung des Felds *duration* konfiguriert wird. In dem Element <*field-validator*> kann mittels dem Property *type* der benötigte Validator angeführt werden. Das Feld *duration* ist ein Pflichtfeld vom Typ Integer, deshalb wird der schon vom Framework zur Verfügung gestellte Validator *requiredint* verwendet. Außerdem muss *duration* im Wertebereich 0 und 31 liegen, was durch die zwei Parameter *min* und *max* spezifiziert wird. Im <*message*>-Element wird die Standard-Fehlermeldung definiert. Somit wird auch das vierte geforderte Kriterium (K 3.4-4) erfüllt.

7.5 Daten-Bindung

Daten-Bindung wird vom Framework verwendet, um Attribute einer JavaBean, die mit den Feldern eines Formulars korrespondieren, automatisch zu setzen. WebWork2 verwendet dazu nicht das Apache Bean-Manipulations-Package (vergleiche Abschnitt 3.5), sondern *OGNL* in Kombination mit dem *ValueStack*. Im nächsten Absatz werden die Vorteile dieser Methode erläutert.

Wie im Abschnitt 7.3 schon erwähnt wurde, wird der *ValueStack* sowohl von der View als auch von den Controllern verwendet, um auf Parameter-Werte zuzugreifen. Durch die dabei verwendete *OGNL* können die Werte zuvor auf den richtigen Typ überprüft werden, bevor sie die Properties der CommandBean befüllen. Somit werden unerwünschte Bind-Exceptions vermieden und die CommandBean-Attribute sind nicht auf Strings beschränkt. Da die zuvor falsch eingegebenen Daten am *ValueStack* liegen, können diese dem/der BenutzerIn als Feedback angezeigt werden, wodurch die Kriterien K 3.5-2 bis K 3.5-4 erfüllt werden.

Soll erreicht werden, dass ein String automatisch in einen bestimmten Typ umgewandelt wird, der nicht standardmäßig von WebWork2 konvertiert werden kann, kann ein eigener Typ-Konverter geschrieben werden, der OGNL's *TypeConverter*-Interface implementieren oder die Klasse *DefaultTypeConverter* erweitern muss. Tritt generell bei der Typumwandlung eine Ausnahme auf, so wird diese vom XWork-*Converter* behandelt. Dieser fügt alle auftretenden Exceptions in eine Map hinzu, die an den *ActionContext* gebunden wird.

Je nach dem, ob der Model-Driven oder der Field-Driven-Ansatz verwendet wird, ist die CommandBean von der Servlet-API und vom Framework unabhängig oder nicht.

Da die Möglichkeit der Unabhängigkeit gegeben ist, wird das Kriterium K 3.5-1 als erfüllt betrachtet.

In der Beispiel-Applikation stellt ein Objekt der Klasse *ScheduleItem* die CommandBean dar. Die Klasse *AddScheduleEntry* implementiert das *Model-Driven*-Interface und stellt die *getObject()*-Methode zur Verfügung, die ein *ScheduleItem*-Objekt zurückgibt, wodurch dieses Objekt an das Formular gebunden wird. Seine Properties bekommen ihre Werte aus den Formularfeldern zugewiesen, ohne dass Code im Controller notwendig ist (vergleiche Abb. 7-4).

7.6 Internationalisierung

In WebWork2 wird Internationalisierung sowohl bei den Fehlermeldungen als auch bei der Gestaltung der Web-Seiten unterstützt. WebWork2 verwendet Property-Dateien um die Beschriftungen und Fehlermeldungen gemäß der jeweiligen Sprache zu definieren. Das Framework liest vom Header des Requests die Orts- und Sprachinformationen aus z.B. Deutsch und Österreich. (Kriterium 3.6-1) Diese Informationen bestimmen die Dateiendung der Property-Datei, so wird zuerst nach einer Datei *<Name>.properties_de_at* gesucht, dann nach *<Name>.properties_de* und wenn beide Dateien nicht verfügbar sind, dann wird die Standard-Property-Datei *<Name>.properties* verwendet. In Abb. 7-10 ist ein Ausschnitt einer Property-Datei zu sehen:

```
input.duration= Duration
input.text= Text
[…]

invalid.duration= You must supply text for this schedule
item
invalid.text= Duration must be between ${min} and ${max}!
[…]
```

Abb. 7-10: Auzug aus AddScheduleEntry.properties

Wie in Abb. 7-10 ersichtlich ist, werden alle Bezeichnungen mit einem bestimmten Schlüssel versehen, durch den der definierte Text erreichbar ist. Die Custom-Tags greifen mit Hilfe dieses Schlüssels auf den Text zu. So kann die richtige Bezeichnung oder Fehlermeldung in der View angezeigt werden. In Abb. 7-11 ist die Verwendung des Internationalisierungs-Konzepts der Custom-Tags dargestellt:

```
<ww:textfield label=“text(‘input.duration’)“
   name=“duration“
   value=’<ww:property value=”duration”/>’ />
```

Abb. 7-11: Verwendung von Internationalisierung

Anstatt bei dem Property *label* direkt Text anzugeben, wird die Funktion *text()* aufgerufen und als Argument der Schlüssel angegeben, mit dem die Bezeichnung in der Property-Datei assoziiert ist. Auch die Validatoren können über den Schlüssel die definierten Fehlermeldungen verwenden. In diesem Fall ist die *text()*-Funktion nicht nötig, der Schlüssel wird durch das Property *key* angegeben. Abb. 7-12 zeigt die Verwendung des Internationalisierungs-Konzepts bei der Validierung.

```
<validators>
  <field name=“duration“>
    <field-validator type=”requiredint”>
    <param name="min">0</param>
    <param name="max">31</param>
    <message key= “invalid.duration”>
      duration must be between ${min} and ${max}!
    </message>
    </field-validator>
  </field>
<validators>
```

Abb. 7-12: Auszug aus AddSchenduleEntry-validation.xml

Wird auf einen Text einer Property-Datei mit einem Schlüssel zugegriffen, wird in folgender Reihenfolge nach dem Property gesucht: *<Action>.properties, <BaseClass>.properties, <Interface>.properties, <package>.properties.* Damit Felder oder Fehlermeldungen, die in mehreren Actions vorkommen, nicht jedes Mal neu definiert werden müssen, kann auch eine *ActionSupport.properties*- Datei verwendet werden.

7.7 Anwendungsspezifisches

Als einziges der anwendungsspezifischen Kriterien wird von WebWork2 das der Datei-Upload-Unterstützung (K 3.7-5) erfüllt. Wie diese Unterstützung von WebWork2 realisiert ist, wird in diesem Abschnitt beschrieben.

7.7.1 Datei-Upload-Unterstützung

WebWork2 bietet integrierte Datei-Upload-Unterstützung. Jeder ankommende Request wird vom Main-Controller auf Formular-Inhalte überprüft, mit denen Dateien geladen werden sollen. Enthält der Request solche Inhalte, erzeugt das Dispatcher-Servlet ein Wrapper-Objekt für den Request. Dieses Wrapper-Objekt heißt *MultiPartRequestWrapper*, mit ihm kann auf die Dateien zugegriffen werden, die geladen werden sollen. Abb. 7-13 zeigt die Verwendung dieser Request-Hülle:

```
MultiPartRequestWrapper multiWrapper
(MultiPartRequestWrapper)ServletActionContext.getRequest();
Enumeration e = multiWrapper.getFileNames();

If (!multiWrapper.hasErrors()) {
  while (e.hasMoreElements()) {
     String val = (String) e.nextElement();
     String contentType = multiWrapper.getContentType(val);
     File file = multiWrapper.getFile(val);
     // Do additional processing/logging...
 }
}
```

Abb. 7-13: Verwendung des MultiPartRequestWrapper

In Abb. 7-13 wird zuerst das gewöhnliche Request-Objekt der Request-Hülle zugewiesen. Dann kann auf diese die Methode *getFileNames()* ausgeführt werden, die gleichzeitig den Upload der Dateien in das definierte Verzeichnis bewirkt. Anschließend wird überprüft, ob dabei Fehler aufgetreten sind. Falls nicht, kann auf die Dateien und deren Inhalts-Informationen durch geeignete *get()*-Methoden zugegriffen werden.

In der *webwork.properties*-Datei, die sich im WEB-INF/classes-Verzeichnis befindet, kann die Upload-Unterstützung konfiguriert werden. In Abb. 7-14 sind die Properties aufgelistet, durch die der Upload entsprechend eingestellt werden kann:

```
webwork.multipart.parser
webwork.multipart.saveDir
webwork.multipart.maxSize
```

Abb. 7-14: Properties zur Upload-Konfiguration

Als *parser* kann jede Klasse gesetzt werden, die die Klasse *MultiPartRequest* erweitert. Wird bei diesem Property nichts angegeben oder wird die angegebene Klasse nicht gefunden, wird die im Framework integrierte Klasse *PellMultiPartRequest* verwendet. Mit *saveDir* kann das Verzeichnis angegeben werden, in das die Dateien gespeichert werden sollen. Standardmäßig wird das Verzeichnis *javax.servlet.context.tempdir* verwendet. *MaxSize* bezeichnet die maximale Größe, die eine Datei haben darf.

7.8 Dokumentation

Dadurch, dass WebWork2 seit der Version 2.0 zwei verschiedene Projekte (WebWork und XWork) umfasst, ist auch die Dokumentation auf zwei verschiedenen Stellen verteilt. Das erschwert die Orientierung in der Dokumentation erheblich. Verwirrend ist außerdem, dass es in beiden Projekten gleiche Dokumentations-Kapitel wie z.B. Validierung gibt, so muss man um einen Überblick über dieses Kapitel zu bekommen an zwei verschiedenen Stellen nachlesen. Trotzdem wird das Kriterium K 3.8-1 erfüllt. Hilfreich sind die JavaDocs (K 3.8-2) und die WebWork2 Wiki-Seite [Webw04w], wo Beispielanwendungen, Tipps und Tricks zur Verfügung stehen. (K 3.8-3)

7.9 Zusammenfassung

Folgende Tabelle soll zum Abschluss der Beschreibung des WebWork2-Frameworks eine zusammenfassende Übersicht über die Eigenschaften von WebWork2 in Bezug auf die Evaluierungskriterien geben:

Controller	Sub-Controller sind JavaBeans	(K 3.1-1)	✓
	Superklassen für zwei verschiedene Kontrollflüsse	(K 3.1-2)	×
	Controller enthalten keine Datenbank-Informationen	(K 3.1-3)	✓
Model	Model ist Servlet- und Framework-unabhängig	(K 3.2-1)	✓
	Model kann mehrere JavaBeans enthalten	(K 3.2-2)	✓
	Model ist durch eigene Komponente realisiert	(K 3.2-3)	✓
View	Verschiedene View-Technologien werden unterstützt	(K 3.3-1)	✓
	View-Austausch ist ohne Controller-Änderung möglich	(K 3.3-2)	✓
Formular-Validierung	Validatoren sind Web-unabhängige JavaBeans	(K 3.4-1)	✓
	Es können eigene Validatoren implementiert werden	(K 3.4-2)	✓
	Deklarative Validierung ist möglich	(K 3.4-3)	✓
	Fehlermeldungen können angezeigt werden	(K 3.4-4)	✓
Daten-Bindung	CommandBean ist Servlet- und Framework-unabh.	(K 3.5-1)	✓
	Typüberprüfungen und Typkonvertierungen	(K 3.5-2)	✓
	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	(K 3.4-3)	✓
	Falsche Daten werden im Formular angezeigt	(K 3.4-4)	✓
Internationalisierung	Länder-Information wird zur Verfügung gestellt	(K 3.6-1)	✓
	Beschriftungen sind vom Java-Code getrennt	(K 3.6-2)	✓
	Fehlermeldungen sind vom Java-Code getrennt	(K 3.6-3)	✓
Anwendungs-spezifisches	JDBC-Abstraktions-Mechanismus	(K 3.8-1)	×
	Deklarative Transaktionsverwaltung ist möglich	(K 3.8-2)	×
	Verwaltung der Abhängigkeiten	(K 3.8-3)	×
	Wizard-Unterstützung	(K 3.8-4)	×
	Datei-Upload-Unterstützung	(K 3.8-5)	✓
Doku-mentation	Referenz-Dokumentation	(K 3.7-1)	✓
	JavaDocs	(K 3.7-2)	✓
	Beispielanwendungen und Tutorials	(K 3.7-3)	✓
Ergebnis	Erfüllte Kriterien:	**22 von 27**	

Tabelle 4: WebWork2 in Bezug auf die Evaluierungskriterien

8. Kapitel

Tapestry

Ein weiteres Open-Source Java-Framework, das zum Jarkata Projekt der Apache Gruppe gehört, ist Tapestry [Tape04]. Es verfolgt – im Gegensatz zu den anderen in dieser Arbeit betrachteten Frameworks - einen *ereignisgesteuerten* Ansatz und „versteckt“ die sonst typischen Konstrukte der Programmierung einer Web-Applikation in seinen Klassen. Dieser Ansatz macht das Framework weitaus umfassender als die anderen in dieser Arbeit beschriebenen Frameworks. Im Folgenden wird Tapestry in seiner momentan aktuellen Version 3.0 beschrieben. Abb. 8-1 zeigt das Zusammenspiel der Komponenten in Tapestry:

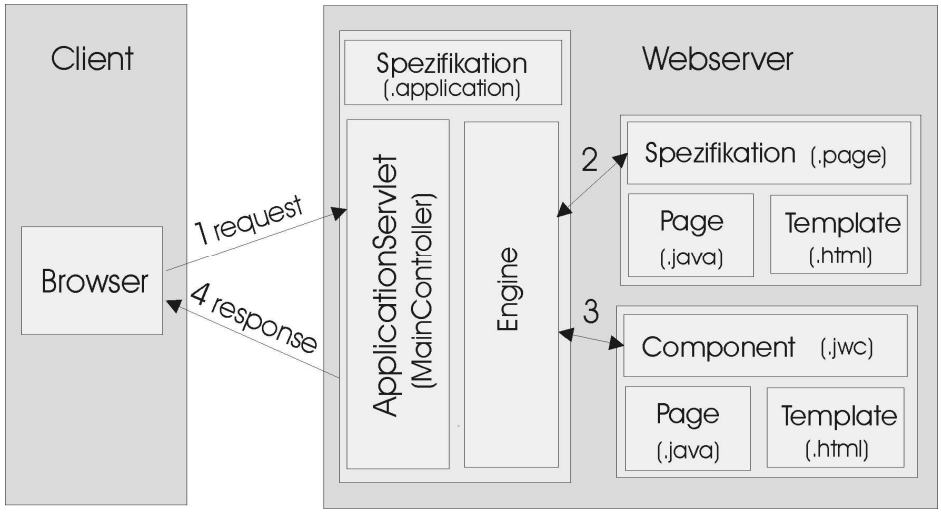

Abb. 8-1: Komponenten von Tapestry

Das *ApplicationServlet*, das den Main-Controller darstellt, liest die Spezifikationsdatei der Anwendung <*NameDerAnwendung*>*.application* und erzeugt die darin

deklarierte *Engine*, die Benutzerdaten kapselt. Der Browser richtet eine Anfrage an die Applikation (1). Diese wird vom *ApplicationServlet* verarbeitet. Das *ApplicationServlet* weiß durch die Spezifikationsdatei, welche Webseiten in dieser Anwendung zur Verfügung stehen und welche Dateien diese beschreiben. Die *Engine* liest die Spezifikationsdatei der benötigten Webseite *<NameDerSeite>.page* (2) und erfährt dadurch von den verwendeten Komponenten, die wiederum in eigenen Spezifikationsdateien beschrieben werden (3). Aus den Templates, der Page und deren Komponenten wird die Webseite zusammengebaut und kann an den Browser zurückgegeben werden (4).

8.1 Controller

Wie auch in allen anderen Frameworks, die in dieser Arbeit beschrieben werden, lagert der Main-Controller, die nicht generischen Controller-Aufgaben an Hilfs-Klassen, so genannte Sub-Controller aus. Deshalb wird die Rolle des Controllers in zwei Abschnitten – Main-Controller und Sub-Controller – beschrieben.

8.1.1 Main-Controller (ApplicationServlet)

Der Eintrittspunkt einer Tapestry-Applikation ist das *ApplikationServlet*, das den Main-Controller darstellt. Dieses Servlet, ist das einzige registrierte Servlet im Deployment-Descriptor und dient als Verbindungsstück zwischen Tapestry und dem Web. Die Klasse, die den Main-Controller realisiert, wird in der *web.xml*-Datei angegeben. In Abb. 8-2 wird diese Deklaration gezeigt:

```
<servlet>
   <servlet-name>tapestry</servlet-name>
  <servlet-class> net.sf.tapestry.ApplicationServlet
   </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>tapestry</servlet-name>
   <url-pattern>tapestry/*</url-pattern>
</servlet-mapping>
```

Abb. 8-2: Auszug aus web.xml

Wie in Abb. 8-2 ersichtlich ist, wird zuerst im Abschnitt *<servlet>* der Name und die Klasse des Main-Controllers angegeben. Anschließend wird im Bereich *<servlet-mapping>* beschrieben, welchem Schema eine URL folgen muss, damit der dazugehörige Request vom Main-Controller verarbeitet wird.

Das *ApplicationServlet* erzeugt für jeden Client eine eigene „Application-Engine“. Das ist ein Framework-Objekt, das die Interaktion mit der eigentlichen Applikation ermöglicht und die Benutzerdaten kapselt. Eine solche „Engine“ implementiert das Interface *IEngine*, das Grundfunktionalitäten für die Seiten und Komponenten der Applikation zur Verfügung stellt. Die „Engine“ wird in der Session gespeichert. Beinahe jede Tapestry-Klasse hat eine *getEngine()*-Methode, die die Instanz der „Engine“ zurückgibt.

Zur Konfiguration der Web-Applikation dient ein XML-Dokument mit der Dateiendung *.application*. In dieser wird angegeben, welche „Engine“ erzeugt werden soll und es wird jeder Web-Seite der Anwendung eine eigene Spezifikations-Datei mit der Dateiendung *.page* zugeordnet. Abb. 8-3 zeigt die Konfigurations-Datei *schedule.application* der Beispiel-Applikation (vergleiche Abschnitt 2.3):

```
<application name="schedule in tapestry"
  engine-class="net.sf.tapestry.engine.SimpleEngine">
  <property name="driverClass">com.mysql.jdbc.Driver
  </property>
  <property name="dbUrl">jdbc:mysql://localhost/schedule
  </property>
  <property name="user">root</property>
  <property name="password">marathon</property>

  <page name="Home" specification-path="/Home.page"/>
  <page name="Add"  specification-path="/Add.page" />

  <component-alias type="SchedTable"
                   specification-path="/SchedTable.jwc"/>
   <library id="contrib" specification-path=
              "/net/sf/tapestry/contrib/Contrib.library"/>
</application>
```

Abb. 8-3: schedule.application

In Abb. 8-3 ist zu sehen, dass die Engine, die für die Beispiel-Applikation verwendet wird, eine Instanz der Klasse *SimpleEngine* ist. Für komplexere Anwendungen kann diese Klasse abgeleitet werden und entweder das Verhalten überschrieben oder zusätzliche Dienste implementiert werden. Nach der Angabe der Engine werden die Attribute definiert, die die Datenbank betreffen. Anschließend werden jeder Seite der Anwendung ein Name und eine Page-Spezifikations-Datei zugeordnet. Jede Web-Applikation benötigt eine „Home-Page“, das ist per Definition die erste Seite. Diese wird automatisch aufgebaut, wenn die Applikation startet. Zum Schluss der Deklaration werden Komponenten und Bibliotheken deklariert, die in der Applikation verwendet werden.

Nachdem die „Engine“ die XML-Konfigurations-Datei gelesen hat, werden die Spezifikations-Datei der angeforderten Seite und das zugehörige Template verwendet, um den Inhalt der Seite zu ermitteln und die Seite aufzubauen.

8.1.2 Sub-Controller (Page)

Jede sichtbare Seite einer Anwendung besteht aus einer Spezifikations-Datei, einem korrespondierenden User-Interface-Template und einer dazugehörigen Klasse. Diese Klasse wird in Tapestry *Page* genannt, kann aber mit einem Sub-Controller eines anderen Frameworks verglichen werden. Deshalb wird in diesem Kapitel über Tapestry der Begriff *Page* gleichgesetzt mit *Sub-Controller* verwendet. Die Page aktiviert Objekte der Geschäftslogik um die Daten zu besorgen, die zur Darstellung notwendig sind.

Der erste Teil eines Sub-Controllers ist die Spezifikation. Diese wird in einer Datei mit dem Namen der Page und der Endung *.page* beschrieben. Abb. 8-4 zeigt die Spezifikation der „Home-Page“ der Beispiel-Applikation:

```
<page-specification class="eval.tapestry.Home">
  <component id="schedTable" type="SchedTable" />
  <component id="Add" type="PageLink">
    <static-binding name="page">Add</static-binding>
  </component>
</page-specification>
```

Abb. 8-4: home.page, (Page-Spezifikation)

In Abb. 8-4 wird im Abschnitt <*page-spezification*> durch das *class* Attribut angegeben, dass die Klasse *Home* die Page darstellt. Diese Seite enthält zwei Komponenten die mit dem <*component*>-Element spezifiziert werden – die Tabelle, in der die vorhandenen Termine aufgelistet werden und einen Hyperlink. Der Link enthält ein <*static-binding*>-Element, das dann eingesetzt wird, wenn ein Attribut nicht von einem Property bereitgestellt wird sondern einen Wert darstellt, der sich nicht ändert. In diesem Fall wird der Name der Page angegeben, zu der der Link führt.

Der nächste Teil der notwendig ist, um in Tapestry eine Seite zu erzeugen ist die zugehörige Java-Klasse, in der Beispiel-Applikation die Klasse *Home*. In Abb. 8-5 wird ein Ausschnitt dieser Klasse dargestellt:

```
public class Home extends BasePage {
  private ScheduleDb scheduleDb;
  private List events;
  public List getEvents() {
    if (scheduleDb == null)
      scheduleDb = createScheduleDb(getEngine());
    return scheduleDb.getEvents();
  }
  private ScheduleDb createScheduleDb(IEngine engine) {
    String dbUrl = engine.getSpecification().
                          getProperty("dbUrl");
    String driverClass = engine.getSpecification().
                                getProperty("driverClass");
    String user = engine.getSpecification().
                          getProperty("user");
    String pw = engine.getSpecification().
                         getProperty("password");
    return new ScheduleDb(driverClass, dbUrl, user, pw);
  }
  protected void firePageBeginRender() {
    super.firePageBeginRender();
    events= scheduleDb.getEvents();
  }
}
```

Abb. 8-5: Auszug aus Home.java (Page-Implementierung)

Die in Abb. 8-5 gezeigte Klasse besitzt eine Liste mit den darzustellenden Termine *events* und eine zugehörige Zugriffsmethode, die dafür zuständig ist die Termine aus der Datenbank zu holen. Die Methode *createScheduleDb()* liest die Datenbank-spezifischen Parameter aus der Spezifikation mit Hilfe der Engine aus. Die Engine steht durch Ableitung der *BasePage*-Klasse mittels *getEngine()* zur Verfügung. Die Methode *firePageBeginRender()* wird aufgerufen um sicher zu stellen, dass die Termin-Anzeige noch mit den in der Datenbank gespeicherten Terminen übereinstimmt. Die Implementierung des Sub-Controllers wird von einer JavaBean übernommen, somit ist das Kriterium K 3.1-1 erfüllt. Da aber in dieser Klasse Datenbank-spezifische Informationen eingelesen werden, ist das Kriterium K 3.1-3 nicht erfüllt.

Es fällt auf, dass in keiner Zeile die beiden Komponenten vorkommen, die in der Spezifikations-Datei definiert wurden. Diese sind selbständige Komponenten. Die Tabelle baut sich selbst auf und behandelt selbst auftretende Ereignisse. Wie jede Tapestry-Komponente benötigt auch sie drei Teile - eine Spezifikations- Datei mit der Endung *.jwc*, ein Template und eine dazugehörende Klasse. Abb. 8-6 zeigt die Spezifikation der Tabellen-Komponente:

```
<component-specification class="eval.tapestry.SchedTable"
        allow-body="no" allow-informal-parameters="yes">
  <component id="table" type="contrib:Table">
    <binding name="tableModel" expression="tableModel"/>
  </component>
</component-specification>
```

Abb. 8-6: Auszug aus schedTable.jwc (Table-Spezifikation)

In Abb. 8-6 wird mit dem Attribut *class* die zur Komponente gehörende Klasse angegeben, die in diesem Fall *SchedTable* ist. In einem <*component*>-Element wird die eigentliche Tabelle angegeben, diese kommt aus der Contrib-Library, die in der Applikations-Spezifikation (vergleiche Abb. 8-3) definiert ist. Um die Tabelle mit Werten zu befüllen, wird ein Model benötigt. Dieses wird mit dem <*binding*>-Element spezifiziert. Zu dem Namen *„tableModel“* gibt es eine korrespondierende Zugriffs-Methode in der *SchedTable*-Klasse, die in Abb. 8-7 dargestellt wird:

```
public class SchedTable extends BaseComponent {
  private List events;
  public ITableModel getTableModel() {
    events = ((Home) getPage()).getEvents();
    SimpleListTableDataModel listTableDataModel =
                  new SimpleListTableDataModel(events);
    return new SimpleTableModel(listTableDataModel,
                                    createColumnModel());
  }
  private ITableColumnModel createColumnModel() {
    String[] col = ScheduleDb.getColumns();
    return new SimpleTableColumnModel(new ITableColumn[] {
                        new StartColumn(col[1]),
                        new DurationColumn(col[2]),
                        new TextColumn(col[3]),
                        new EventTypeColumn(col[4])});
  }
}
private class StartColumn extends SimpleTableColumn {
    public StartColumn(String colName) {
      super(colName);
    }
    public Object getColumnValue(Object row) {
      return ((ScheduleItem) row).getStart();
    }
  }
  […] //DurationColumn, TextColumn, EventTypeColumn
}
```

Abb. 8-7: Auszug aus SchedTable.java (Table-Implementierung)

Wie in Abb. 8-7 ersichtlich ist, erweitert *SchedTable* die Klasse *BaseComponent,* die Basis-Klasse aller Komponenten in Tapestry. In der Methode *getTableModel()* holt sich die Komponente die anzuzeigenden Termine aus der *Home*-Page. Die Klasse *SimpleListTableDataModel* macht aus der Liste der vorhandenen Termine ein „Data-Model“, das gemeinsam mit dem „Column-Model“ ein „Table-Model“ ergibt, das die Tabellen-Komponente zur Anzeige benötigt (vergleiche Abb. 8-6).

Das „Data-Model“ repräsentiert die Zeilen der Tabelle, das heißt, jeder Termin steht in einer Zeile der Tabelle. Das „Column-Model“ stellt die Spalten der Tabelle dar. Die Tabelle hat vier Spalten zusätzlich zu der Spalte, in der die Termine stehen. Es wird zur Erzeugung des „Column-Model“ für jede dieser Spalten ein eigenes Objekt benötigt, das diese repräsentiert. Deshalb werden für die vier Attribute der Klasse *ScheduleItem* (*start*, *duration*, *text* und *eventType*) vier private Klassen implementiert.

8.2 Model (Visit Object)

Damit Daten von einer Web-Seite zur anderen transportiert werden können, stellt Tapestry den Visit-Objekt-Mechanismus zur Verfügung. Das Visit-Objekt ist in der „Engine“ enthalten, die eine Zugriffs- und eine Änderungs-Methode dafür zur Verfügung stellt. Als Visit-Objekt kann jedes beliebige Objekt verwendet werden, es ist nicht nötig ein bestimmtes Interface zu implementieren bzw. von einer Basis-Klasse abzuleiten. Somit wird das Kriterium K 3.2-1 erfüllt, da das Model Framework- und Servlet-API-unabhängig ist. Es könnte auch eine Map als Visit-Objekt verwendet werden, es müsste dann kein Wrapper-Objekt erstellt werden, damit das Model mehrere JavaBeans der Geschäftslogik enthalten kann. Dadurch wird die Erfüllung des Kriteriums K 3.2-2 möglich gemacht. Das Visit-Objekt wird in der Applikations-Spezifikation als Property angegeben, und kann so automatisch, mit Hilfe des Standard-Konstruktors erzeugt werden.

Um Informationen von einer Komponente einer Seite zur anderen zu transportieren, kann nicht nur das Visit-Objekt verwendet werden, sondern auch folgende andere Methode. Da jede Komponente mittels der *getPage()*-Methode ihre Seite erreicht, kann die Komponente direkt auf die darzustellenden Daten, die die Page zur Verfügung stellt, zugreifen. Diese Methode wird in der Beispiel-Applikation angewandt. In Abb. 8-8 wird noch einmal die Methode *getTableModel()* der *SchedTable*-Komponente (vergleiche Abb. 8-7) dargestellt um den Zugriff auf die *Home*-Komponente zu verdeutlichen:

```
public ITableModel getTableModel() {
   events = ((Home) getPage()).getEvents();
   SimpleListTableDataModel listTableDataModel =
                new SimpleListTableDataModel(events);
  return new SimpleTableModel(listTableDataModel,
                              createColumnModel());
}
```

Abb. 8-8: Direkter Zugriff auf die Model-Daten der Page

Die in Abb. 8-8 gezeigte Methode *getTableModel()* kann mit Hilfe der *getPage()*-Methode, die von der Klasse *BaseComponent* geerbt wurde, auf die *Home*-Page zugreifen und sich die nötigen Informationen über die bestehenden Termine beschaffen. Diese Vorgangsweise ist vergleichbar mit dem Controller-As-Model-Pattern (vergleiche Kapitel 6.2). Es gibt daher in Tapestry für das Model die Möglichkeit in einem eigenen Objekt verwirklicht zu werden oder im Controller enthalten zu sein. Da eine eigene Komponente möglich ist, wird das Kriterium K 3.2-3 als erfüllt betrachtet.

8.3 View

Zur Präsentation der Daten verwendet Tapestry eine eigene proprietäre Alternative zu Scripting-Sprachen wie JSP [Sun03a] und Velocity [Velo04]. Damit wird eine Komponenten basierte Darstellung der Webseiten erreicht. Die Präsentation der Anwendung ist deshalb sehr flexibel, da sie aus beliebigen Komponenten zusammengebaut wird. So kann nicht nur die ganze View einfach geändert werden, ohne dass Controller-Code-Änderungen notwendig sind, sondern auch einzelne Komponenten können leicht gegen andere ausgetauscht werden. Tapestry wird dem Kriterium K 3.3-2 dadurch mehr als gerecht.

In Tapestry ist es nicht möglich, andere View-Technologien einzusetzen, deshalb ist das Kriterium K 3.3-1 nicht erfüllt. In den Templates dienen gewöhnliche HTML-Elemente als Platzhalter. Diese Platzhalter werden dann von den Komponenten und JavaBeans ersetzt, die in der *<Name>.page* - Spezifikationsdatei angeführt sind. Innerhalb der HTML-Elemente wird die Object *Graph Navigation Language OGNL* verwendet (vergleiche Abschnitt 7.3), um auf das darunter liegenden Model zuzugreifen. Die Vorteile der *OGNL*-Verwendung zeigen sich in der Formular-Validierung und der Daten-Bindung (siehe Abschnitte 8.4 und 8.5).

In Abb. 8-9 ist der dritte Teil abgebildet, der neben der Spezifikation und der Java-Klasse notwendig ist, um eine Web-Site zu erstellen, das Template der *Home*-Page:

```
<html>
  <head><title>Schedule</title></head>
  <body>
    <h2>Schedule</h2>
    <span jwcid="schedTable"></span><p>
    <a jwcid="Add">Add a new Schedule Item</a>
  </body>
</html>
```

Abb. 8-9: home.html (Home-Template)

Es kann jede Komponente als Platzhalter eingesetzt werden, es muss nur ein Attribut *jwcid* angegeben werden. Dann ersetzt Tapestry dieses HTML-Element mit der Komponente, die in der Spezifikation angegeben wurde. Im obigen Beispiel wird zuerst ein <*span*>-Element als Platzhalter eingesetzt, die *jwcid „schedTable“* stimmt mit der *id* der ersten in der Page-Spezifikation angegebenen Komponente überein (vergleiche Abb. 8-4). Somit wird das Element mit einer Komponente vom Typ *SchedTable* ersetzt. Auch das *jwcid*-Attribut des <*a*>-Elements referenziert die spezifizierte Link-Komponente. Abb. 8-10 zeigt das Template der *SchedTable*-Komponente:

```
<table border="1" jwcid="table" />
```

Abb. 8-10: schedTable.html (Table-Template)

Das in Abb. 8-10 dargestellte, zur Tabelle gehörende Template ist sehr einfach. Es beinhaltet nur ein HTML-Element, das wieder durch das *jwcid*-Attribut einer spezifizierten Komponente zugeordnet ist.

8.4 Validierung

Zur Validierung sind in Tapestry vier verschiedene Komponenten notwendig. *FieldLabel* und *ValidField* sind für die Präsentation der Benutzerschnittstelle verantwortlich, den Kern der Validierung übernehmen *Validatoren* und deren so genannte *Delegates*.

Die Komponente *ValidField* ist das Tapestry-Äquivalent zum Standard-HTML Textfeld-Element, sie repräsentiert ein Eingabefeld in einem Formular. Sie kann für alle Felder eines Formulars eingesetzt werden, egal welchen Typ die Eingabe haben soll. In Abb. 8-11 ist ein Auszug der Spezifikations-Datei der *Add* Page dargestellt, in dem die *ValidField*–Komponente spezifiziert ist, mit der der Wert für das Attribut *duration* eingegeben werden kann:

```
<component id="duration" type="ValidField">
  <static-binding name="displayName" value="Duration"/>
  <binding name="validator" session="beans.intValidator"/>
  <binding name="value" expression=
                                  “scheduleItem.duration"/>
</component>
```

Abb. 8-11: Auszug aus Add.page (Deklaration des ValidFields)

Die in Abb. 8-11 spezifizierte Komponente *duration* ist vom Typ *ValidField*. In dem Element <*static-binding*> wird ihr ein *displayName* zugewiesen. Dieser Name wird auch vom Validator für die Generierung von Fehlermeldungen verwendet. Mit *validator* wird dem *ValidField* ein Validator zugeordnet, der im Bereich der Bean-Spezifikation der *Add.page*-Datei definiert ist und der die eindeutige id *intValidator* hat. Abb. 8-12 zeigt den Ausschnitt der *Add.page*, in dem dieser Validator spezifiziert wird:

```
<bean  name="intValidator"
       class="org.apache.tapestry.valid.IntValidator"
       lifecycle=”page”>
  <set-property name="minimum" expression="1"/>
  <set-property name="maximum" expression="30"/>
</bean>
```

Abb. 8-12: Auszug aus Add.page (Deklaration des Validators)

In Abb. 8-12 ist ersichtlich, wie der Validator für die Integer-Variable *duration* deklariert und konfiguriert wird. Die Klasse *IntValidator* wird, wie viele andere Standard-Validatoren, vom Framework zur Verfügung gestellt. Es können die Properties *minimum* und *maximum* gesetzt werden. Tapestry unterstützt also deklarative Validierung und wird dem Evaluierungskriterium K 3.4-3 gerecht.

Validatoren sind in Tapestry dafür zuständig, Benutzereingaben in Objekte zu konvertieren und verschiedene Überprüfungen vorzunehmen. Sie sind Framework- und Servlet-API-unabhängige JavaBeans (Kriterium K 3.4-1) und implementieren das Interface *IValidator*, das die Methoden *toString()* und *toObject()* verlangt. In den meisten Fällen ist es nicht notwendig, eigene Validatoren zu implementieren, da Tapestry schon Standard-Validatoren zur Verfügung stellt. Es gibt zum Beispiel den *StringValidator*, der konfiguriert werden kann, wie viele Zeichen eingegeben werden müssen, oder den *NumberValidator*, der für alle numerischen Typen angewandt werden kann und dem ein Minimum und ein Maximum-Wert angegeben werden kann. Es gibt auch einen *DateValidator*, bei dem ebenfalls Minimum und Maximum spezifiziert werden können.

Für den Fall, dass ein Validator benötigt wird, der nicht vom Framework zur Verfügung gestellt wird, bietet Tapestry *BasisValidator* als Basisklasse für benutzerdefinierte Validatoren an. Diese Klasse enthält die Methode *checkRequired()*, die *true* zurückgibt wenn der Wert null oder leer ist und wirft eine *ValidationException* wenn das Property *required* des Validators auf *true* gesetzt ist. Das Kriterium K 3.4-2 wird durch diese Möglichkeit der Erweiterung erfüllt.

FieldLabel ist die Partner-Komponente von *ValidField*. Jedes *FieldLabel* ist durch den Parameter *field* mit dem *ValidField* verbunden. Es kann die Ausgabe des Feldes danach richten, ob ein Fehler aufgetreten ist oder nicht. Kriterium K 3.4-4 wird dadurch erfüllt. Zum Beispiel kann es den Wert in roter Farbe ausgeben. Außerdem hat es Zugriff auf den *displayName* Parameter des *ValidFields*, wodurch sichergestellt wird, dass die Beschriftung des Feldes mit der in den Fehlermeldungen vorkommenden Feld-Bezeichnung übereinstimmt. Abb. 8-13 zeigt ein Beispiel für die Verwendung der *FieldLabel*- und der *ValidField*- Komponenten:

```
<table>
  <tr><td>
    <span jwcid="@FieldLabel"
          field="ognl:components.duration"/>
  </td><td>
    <input jwcid="duration" type="text" size="4"/>
  </td></tr>
</table>
```

Abb. 8-13: Auszug aus Add.html (FieldLabel und ValidField)

Wie in Abb. 8-13 ersichtlich ist, wird die Beschriftung des Eingabefeldes mit der Komponente *FieldLabel* realisiert. Der Klammeraffe vor dem Komponentennamen bedeutet, dass die Komponente vom Framework zur Verfügung gestellt wird. Diese ist durch den *OGNL*-Ausdruck *components.duration* mit dem *ValidField duration* verknüpft, und hat somit Zugriff auf dessen Attribut *displayName*. Der Wert dieses Attributs wird auf der Webseite angezeigt. Die Komponente *duration* vom Typ *ValidField* stellt das Eingabefeld dar und ist mit dem Property des *ScheduleItem duration* verbunden (vergleiche Abb. 8-11).

Delegates sind vom Framework zur Validierung benutzte Objekte. Sie werden dazu verwendet, um die Zuordnung des fehlerhaften Feldes, seinen Wert und die dazugehörige Fehlermeldung der Page und dem Template zugänglich zu machen. Ein solches Delegate wird einem Formular zugeordnet und verwaltet die Fehler von allen seinen Feldern. Validatoren teilen den *ValidFields* mittels Ausnahmen mit, ob und

welche Fehler aufgetreten sind. Abb. 8-14 zeigt die Verwendung von Delegates am Beispiel des Templates der Seite *Add*.

```
<span jwcid="@Conditional"
  condition=" ognl:beans.delegate.hasErrors">
  <table class="error">
    <tr valign="top"><td>
      <span jwcid="@Delegator"
        delegate="ognl:beans.delegate.firstError">
        Error Message
      </span>
  </td></tr></table>
</span>
```

Abb. 8-14: Auzug aus Add.html (Delegates zur Fehleranzeige)

In dem in Abb. 8-14 dargestellten HTML-Code wird eine Konditional-Komponente, die von Tapestry zur Verfügung gestellt wird, verwendet, damit die Fehlermeldung nur angezeigt wird, wenn Fehler aufgetreten sind. Mit dem *OGNL*-Ausdruck *beans.delegate* kann auf das Delegate-Objekt zugegriffen werden und seine Methode *hasErrors* abgefragt werden. Im Fehlerfall wird der erste Fehler, der im Delegate-Objekt gespeichert ist, angezeigt.

8.5 Daten-Bindung

Auch in Tapestry gibt es die Möglichkeit der Daten-Bindung, dazu muss die CommandBean und ihre Zugriffs- und Änderungs-Methode in der Page enthalten sein. Es kann jedoch jede beliebige JavaBean als CommandBean verwendet werden, deshalb wird das Kriterium K 3.5-1 erfüllt. Dann kann die in der Page verwendete Komponente, z.B. ein Eingabefeld, mit einem Property der CommandBean verbunden werden. Abb. 8-15 zeigt noch einmal diese Bindung (vergleiche Abb. 8-11):

```
<component id="start" type="ValidField">
  <static-binding name="displayName" value="Start Date"/>
  <binding name="value"
           expression="scheduleItem.start"/>
</component>
```

Abb. 8-15: Auszug aus Add.page

In dem in Abb. 8-15 dargestellten Beispiel wird die Komponente *start* definiert. In dem <*binding*>-Element wird ihr das Property *scheduleItem.start* zugeordnet, was bedeutet, dass es in der Page ein Property *ScheduleItem* mit geeigneter Zugriffs- und Änderungs-Methode geben muss, das wiederum sowohl eine *getStart()*- als auch eine

setStart()-Methode enthalten muss. Die Komponente *ValidField* kümmert sich um die Typüberprüfung und führt eine eventuell notwendige Konvertierung durch. Kriterium K 3.5-2 wird dadurch erfüllt. Wie im Abschnitt 8.4 schon erwähnt, werden Fehler von einer Komponente z.B. *ValidField* zur anderen z.B. *FieldLabel* kommuniziert. Es steht also ein ausgereifter Ausnahme-Behandlungs-Mechanismus zur Verfügung, somit wird Tapestry auch dem Kriterium K 3.5.3 gerecht.

Da fehlerhafte Eingaben vom Framework in einem Delegate-Objekt gespeichert werden (vergleiche Kapitel 8.4), können diese dem Benutzer/ der Benutzerin erneut angezeigt werden, womit das Kriterium K 3.5-4 erfüllt wird.

8.6 Internationalisierung

In Tapestry können entweder einzelne Komponenten oder ganze Templates an die Ort- bzw. Sprachgegebenheiten der BenutzerInnen angepasst werden. Das Kriterium K 3.6-1 wird dadurch erfüllt. Jede Komponente kann die benötigte Anzahl von Beschriftungen und Fehlermeldungen in einer *[Komponentenname].properties*-Datei ablegen. Zum Beispiel wird bei Bearbeitung der Komponente *Add.page* für die deutsche Sprache in Österreich in folgender Reihenfolge nach einer Property-Datei gesucht: *Add_de_at.properties, Add_de.properties* und *Add.properties*. Mit dem Element <*string-binding*> haben Komponenten Zugriff auf die Beschriftungen und Fehlermeldungen, die in diesen Dateien definiert sind. Das bedeutet, dass die Kriterien K 3.6-2 und K 3.6-3 erfüllt sind. Es ist auch möglich das gesamte Template nach lokalen Einstellungen auszurichten. Auch bei dieser Variante der Internationalisierung wird an den Dateinamen die Orts- und Sprachinformation angehängt. So wird für die Anzeige der Komponente *Add.page* in folgender Reihenfolge nach dem dazugehörigen Template gesucht: *Add_de_at.html, Add_de.html* und *Add.html*.

8.7 Anwendungsspezifisches

Genau wie Struts und WebWork2 erfüllt Tapestry als einziges von den anwendungsspezifischen Evaluierungs-Kriterien die Forderung nach einer Datei-Upload-Unterstützung. In diesem Abschnitt wird die Realisierung dieser Unterstützung näher beschrieben.

8.7.1 Datei-Upload-Unterstützung

Tapestry bietet schon eine fertige Upload-Komponente an, die nur angewandt werden muss. Abb. 8-16 zeigt die Anwendung dieser Komponente in einem Formular eines Templates:

```
<form jwcid="@Form" listener="ognl:listener.formSubmit">
  <table><tr><td>
       File: 
       <input jwcid="@Upload" file="ognl:file"
                                type= "file"/>
    </td></tr>
    <tr><td>
      <input type="submit" value="Upload"/>td>
  </tr></table>
</form>
```

Abb. 8-16: Formular im Upload-Template

In Abb. 8-16 wird im Template durch das *jwcid*-Attribut auf die Upload-Komponente verwiesen, diese ersetzt bei der Ausführung das <*input*>-Element. Durch den *OGNL*-Ausdruck wird die zu ladende Datei an das Attribut *file* der Page gebunden und so automatisch gesetzt. In Abb. 8-17 ist eine Page dargestellt, die die geladene Datei verwendet:

```
public class UploadPage extends BasePage {

  public static final Format SIZE_FORMAT =
                        NumberFormat.getNumberInstance();
  private IUploadFile file;

  public IUploadFile getFile() {return file;}
  public void setFile(IUploadFile value) {file = value;}
  public void formSubmit(IRequestCycle cycle) {
    InputStream fis = file.getStream();
    FileOutputStream fos = new FileOutputStream(new
                              File(file.getFileName()));
    byte[] buffer = new byte[1024];
    do {
      int length = fis.read(buffer);
      fos.write(buffer, 0, length);
   }while(length > 0);
      fis.close();  fos.close();
   }
}
```

Abb. 8-17: Upload-Page

In Abb. 8-17 ist zu sehen, dass die von der Upload-Komponente auf den Server geladene Datei durch ein Objekt vom Typ *IUploadFile* repräsentiert wird. Die Page enthält ein dafür vorgesehenes Property *file* mit entsprechender Änderungs-Methode. Das Property wird automatisch gesetzt und kann in der *formSubmit()*-Methode der Page verarbeitet werden.

8.8 Dokumentation

Tapestry hat eine außerordentlich gute und umfassende Dokumentation, sie umfasst folgende Abschnitte: Component-Reference, API-Documentaion, Developer-Guide, User-Guide und Contributor's Guide. Die ganze Dokumentation ist mit vielen Beispielen zum besseren Verständnis versehen. Außerdem gibt es eine hilfreiche Wiki-Seite [Tape04w] mit Tipps und Tricks sowie FAQs und einem Whiteboard. Somit sind die Kriterien K 3.8-1 bis K 3.8-3 erfüllt. Als zusätzliche Literatur ist das Buch *Tapestry in Action* von Howard M. Lewis Ship [Howa04] erhältlich.

8.9 Zusammenfassung

Folgende Tabelle soll zum Abschluss der Beschreibung des Tapestry-Frameworks eine zusammenfassende Übersicht über die Eigenschaften von Tapestry in Bezug auf die Evaluierungskriterien geben:

Controller	Sub-Controller sind JavaBeans	(K 3.1-1)	✓
	Superklassen für zwei verschiedene Kontrollflüsse	(K 3.1-2)	✗
	Controller enthalten keine Datenbank-Informationen	(K 3.1-3)	✗
Model	Model ist Servlet- und Framework-unabhängig	(K 3.2-1)	✓
	Model kann mehrere JavaBeans enthalten	(K 3.2-2)	✓
	Model ist durch eigene Komponente realisiert	(K 3.2-3)	✓
View	Verschiedene View-Technologien werden unterstützt	(K 3.3-1)	✗
	View-Austausch ist ohne Controller-Änderung möglich	(K 3.3-2)	✓
Formular-Validierung	Validatoren sind Web-unabhängige JavaBeans	(K 3.4-1)	✓
	Es können eigene Validatoren implementiert werden	(K 3.4-2)	✓
	Deklarative Validierung ist möglich	(K 3.4-3)	✓
	Fehlermeldungen können angezeigt werden	(K 3.4-4)	✓
Daten-Bindung	CommandBean ist Servlet- und Framework-unabh.	(K 3.5-1)	✓
	Typüberprüfungen und Typkonvertierungen	(K 3.5-2)	✓
	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	(K 3.4-3)	✓
	Falsche Daten werden im Formular angezeigt	(K 3.4-4)	✓
Internationalisierung	Länder-Information wird zur Verfügung gestellt	(K 3.6-1)	✓
	Beschriftungen sind vom Java-Code getrennt	(K 3.6-2)	✓
	Fehlermeldungen sind vom Java-Code getrennt	(K 3.6-3)	✓
Anwendungs-spezifisches	JDBC-Abstraktions-Mechanismus	(K 3.8-1)	✗
	Deklarative Transaktions-Verwaltung ist möglich	(K 3.8-2)	✗
	Verwaltung der Abhängigkeiten	(K 3.8-3)	✗
	Wizard-Unterstützung	(K 3.8-4)	✗
	Datei-Upload-Unterstützung	(K 3.8-5)	✓
Doku-mentation	Referenz-Dokumentation	(K 3.7-1)	✓
	JavaDocs	(K 3.7-2)	✓
	Beispielanwendungen und Tutorials	(K 3.7-3)	✓
Ergebnis	Erfüllte Kriterien:	**20 von 27**	

Tabelle 5: Tapestry in Bezug auf die Evaluierungskriterien

9. Kapitel

Zusammenfassung der Evaluierungs-ergebnisse

In diesem Kapitel werden zur besseren Übersicht zuerst die fünf Frameworks an Hand der Ergebnisse der Kategorien der einzelnen Evaluierungskriterien einander gegenüber gestellt. Diese Gegenüberstellung erfolgt sowohl in tabellarischer Form als auch auf Basis kurzer textueller Zusammenfassungen. Anschließend werden besondere Vorzüge und Nachteile jedes Frameworks noch einmal kurz diskutiert.

9.1 Controller

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.1-1	Sub-Controller sind JavaBeans	✓	✓	✓	✓	✓
K 3.1-2	Superklassen für zwei verschiedene Kontrollflüsse	×	✓	×	×	×
K 3.1-3	Controller enthalten keine Datenbank-Informationen	×	✓	✓	×	×

Tabelle 6: Vergleich in Bezug auf die Controller-Kriterien

Alle Frameworks verwenden Servlets für den Main-Controller und konfigurierbare JavaBeans für die Rolle des Sub-Controllers.

Nur Spring bietet Superklassen an, die die gesonderte Art des Kontrollfluss bei Formularen realisieren. Es wird somit als einziges Framework allen Controller-Kriterien gerecht.

Eine saubere Schichtentrennung in Bezug auf Datenbank bezogene Informationen unterstützen Spring und WebWork2 durch die Realisierung die externe Verwaltung der Abhängigkeiten von Java-Klassen („Inversion of Control“).

Struts und Maverick erfüllen lediglich die Forderung nach JavaBeans als Sub-Controller.

9.2 Model

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.2-1	Model ist Servlet und Framework unabhängig	×	✓	✓	✓	✓
K 3.2-2	Model kann mehrere JavaBeans enthalten	✓	✓	✓	✓	✓
K 3.2-3	Model ist durch eigene Komponente realisiert	✓	✓	✓	✓	✓

Tabelle 7: Vergleich in Bezug auf die Model-Kriterien

Struts ist das einzige Framework, das sein Model vom Framework und der Servlet-API abhängig macht. Das hat den Nachteil, dass das Model nicht direkt an die Geschäftslogik übergeben werden kann, sondern dafür ein Wrapper-Objekt implementiert werden muss, um eine saubere Schichtentrennung zu wahren. Die Models aller anderen Frameworks sind Framework- und Servlet-API-unabhängig.

Obwohl in allen Frameworks bis auf Struts eine Map als Model eingesetzt werden kann, um mehrere JavaBeans in ein Model zusammenzufassen, bietet Spring für dieses Konzept schon besondere Unterstützung.

Alle Frameworks kommen der Forderung nach einer eigenen Komponente für das Model nach, obwohl Maverick und WebWork2 auch die andere Variante des „Model-As-Controller-Pattern“ zur Verfügung stellen.

9.3 View

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.3-1	Verschiedene View-Technologien werden unterstützt	✓	✓	✓	✓	×
K 3.3-2	View-Austausch ist ohne Controller-Änderung möglich	✓	✓	✓	✓	✓

Tabelle 8: Vergleich in Bezug auf die View-Kriterien

Tapestry benötigt seine eigene Scripting-Sprache um den ereignisgesteuerten Kontrollfluss gemeinsam mit einer komponentenbasierten Arbeitsweise umzusetzen. Alle anderen, hier verglichenen Frameworks sind ziemlich flexibel in Bezug auf die verwendete View-Technologie. Durch die Umsetzung des MVC-Design-Patterns ist der Austausch der View ohne Controller-Änderungen in allen Frameworks möglich.

Struts, Spring und WebWork2 stellen eine Reihe an Custom-Tags zur Verfügung. Dabei muss erwähnt werden, dass die Custom-Tags von Struts durch ein Plug-In in der Entwicklungsumgebung Dreamweaver direkt von den Web-Designern verwendet werden können.

9.4 Validierung

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.4-1	Validatoren sind Web-unabhängige JavaBeans	×	✓	✓	×	✓
K 3.4-2	Es können eigene Validatoren implementiert werden	×	✓	✓	×	✓
K 3.4-3	Deklarative Validierung ist möglich	✓	×	✓	×	✓
K 3.4-4	Fehlermeldungen können angezeigt werden	✓	✓	✓	×	✓

Tabelle 9: Vergleich in Bezug auf die Validierungs-Kriterien

Im Kontext der Validierungskriterien schneiden WebWork2 und Tapestry am besten ab, denn diese beiden Frameworks erfüllen alle geforderten Kriterien in diesem

Bereich. Maverick unterstützt Validierung überhaupt nicht, was sicher ein großer Nachteil gegenüber den anderen Frameworks bedeutet.

Spring, WebWork2 und Tapestry bieten ein Interface zur Erstellung eigener Validatoren an. Da Struts durchgehend eher auf konkreten Klassen als auf Interfaces aufgebaut ist, sind seine Validatoren vom Framework und der Servlet-API abhängig, was ein weiteres Manko für dieses Framework bedeutet.

Die Flexibilität sowohl der Präsentation als auch der Wiederverwendbarkeit von Formularen wird erheblich gesteigert, wenn Validierungsregeln und Fehlermeldungen außerhalb des Java-Codes verwaltet werden. Deshalb unterstützen alle Frameworks (außer Maverick) dieses Konzept der deklarativen Validierung und bieten die Möglichkeit, die Validierung in XML-Dateien zu konfigurieren. Struts, WebWork2 und Tapestry stellen zusätzlich Standard-Validatoren zur Verfügung, die in Spring selbst implementiert werden müssen.

9.5 Daten-Bindung

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.5-1	CommandBean ist Servlet und Framework unabhängig	×	✓	✓	✓	✓
K 3.5-2	Typüberprüfungen und Typkonvertierungen	×	✓	✓	×	✓
K 3.5-3	Ausnahmebehandlung bei Daten-Bindungs-Fehlern	×	✓	✓	×	✓
K 3.5-4	Falsche Daten werden im Formular angezeigt	×	✓	✓	×	✓

Tabelle 10: Vergleich in Bezug auf die Daten-Bindungs-Kriterien

Einen großen Minuspunkt für Struts und Maverick stellt die Verwendung des Apache *BeanUtil*-Packages im Zuge der Daten-Bindung dar. Der nicht überzeugende Fehler-Mechanismus, der keinen Platz vorsieht, um die falsch eingegeben Daten zu speichern, sofern sie nicht vom Typ String sind, und die fehlende Ausnahme-Behandlungs-Möglichkeit sind Negativ-Aspekte dieses Packages.

WebWork2 und Tapestry verwenden die *Object Graph Navigation Language* in Kombination mit dem *OGNL ValueStack* als überzeugende Alternative für das

BeanUtil-Package. Spring hingegen liefert seine eigene Implementierung, die ebenfalls keine der *BeanUtil*-Probleme aufweist.

9.6 Internationalisierung

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.6-1	Länder-Information wird zur Verfügung gestellt	✓	✓	✓	✓	✓
K 3.6-2	Beschriftungen sind vom Java-Code getrennt	✓	✓	✓	✓	✓
K 3.6-3	Fehlermeldungen sind vom Java-Code getrennt	✓	✓	✓	×	✓

Tabelle 11: Vergleich in Bezug auf Internationalisierungs-Kriterien

Da Maverick Validierung und somit die Verwaltung von Fehlermeldungen nicht unterstützt, schneidet es als schlechtestes der Frameworks ab. Außerdem können nur gesamte Webseiten gemäß den Orts- und Sprachgegebenheiten angezeigt werden, was einen sehr hohen Änderungsaufwand mit sich zieht, da jede Seite in mehrfacher Ausführung (für jedes Land/ für jede Sprache) vorhanden ist.

In Tapestry können sowohl einzelne Komponenten als auch gesamte Templates international angepasst werden, die Verbindung zu den in Property-Dateien abgelegten Beschriftungen und Fehlermeldungen wird in der Spezifikations-Datei angegeben. Struts, Spring und WebWork2 realisieren diese Verbindung in Custom-Tags.

Spring bietet zusätzlich zu der – von den anderen Frameworks ausschließlich verwendeten – Möglichkeit über den Header des Request an die Orts- bzw. Sprachinformation der BenutzerInnen heranzukommen, auch die Optionen die Information aus der Session bzw. aus einem Cookie auszulesen.

9.7 Anwengunsspezifisches

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.7-1	JDBC-Abstraktions-Mechanismus	×	✓	×	×	×
K 3.7-2	Deklarative Transaktions-Verwaltung ist möglich	×	✓	×	×	×
K 3.7-3	Verwaltung der Abhängigkeiten	×	✓	×	×	×
K 3.7-4	Wizard-Unterstützung	×	✓	×	×	×
K 3.7-5	Datei-Upload-Unterstützung	✓	✓	✓	×	✓

Tabelle 12: Vergleich in Bezug auf anwendungsspezifische Kriterien

Alle Frameworks bieten Hilfestellung beim Datei-Upload außer Maverick. Dabei liefert Tapestry die einfachste Lösung, da es schon eine fertige Upload-Komponente beinhaltet. Spring bietet als einziges Framework Unterstützung bei Wizard-Formularen, was ein großes Plus für Spring darstellt, da der sonst aufwändige Umgang mit mehrseitigen Formularen durch die automatische Weiterleitung und die Information von welcher Seite die Eingaben stammen erheblich vereinfacht wird.

Die restlichen anwendungsspezifischen Kriterien beziehen sich nicht auf ein Web-Framework, werden aber alle von Spring erfüllt. Diese sind als Bonus-Punkte für Spring aufzufassen.

9.8 Dokumentation

		Struts	Spring	WebWork2	Maverick	Tapestry
K 3.8-1	Referenz-Dokumentation	✓	×	✓	×	✓
K 3.8-2	JavaDocs	✓	✓	✓	×	✓
K 3.8-3	Beispielanwendungen und Tutorials	✓	✓	✓	✓	✓

Tabelle 13: Vergleich in Bezug auf die Dokumentations-Kriterien

Spring und Maverick schneiden im Bereich der Dokumentation schlecht ab. Beispielanwendungen und Tutorials werden jedoch von allen Frameworks zur Verfügung gestellt.

9.9 Pros und Contras der einzelnen Frameworks

9.9.1 Struts

Struts erleichtert es erheblich, eine Web-Applikation nach dem „Model 2"-Konzept zu entwickeln. Es wird der Aufwand zur Validierung von Standardtypen durch die Möglichkeit der deklarativen Validierung auf ein Minimum reduziert, was den größten Pluspunkt von Struts darstellt. Weiters ist positiv zu vermerken, dass die von Struts zur Verfügung gestellten Tags von der Entwicklungsumgebung Dreamweaver durch ein Plug-In wie gewöhnliche HTML-Tags verwendet werden können. Weniger hervorragend ist Struts in Bezug auf seine Architektur, da Struts fast ausschließlich auf konkreten Klassen basiert. So ist es zum Beispiel nicht möglich, eigene Validatoren in das Framework zu integrieren. Die Vorgehensweise, Daten von einem *ActionForm*-Objekt in ein anderes Objekt zu kopieren, um sie an die Geschäftslogik übergeben zu können, ist äußerst unelegant und wird auch von keinem der anderen Frameworks erzwungen. Zusätzlich schneidet Struts sehr schlecht in der Kategorie der Daten-Bindung ab, einerseits auf Grund der Servlet- und Framework-Abhängigkeit der CommandBean und andererseits wegen der Verwendung des Jarcata *BeanUtil*-Package.

9.9.2 Spring

Spring basiert auf der Verwendung von Interfaces und dem „Inversion of Control"-Entwurfsmuster, was zu einer losen Kopplung der Komponenten führt und von einer sauber durchdachten Architektur zeugt [John03]. Die Funktionalität, die Spring von den anderen Frameworks unterscheidet ist die Unterstützung bei so genannten „Wizard"-Formularen. Durch eine praktische Superklasse wird der Kontrollfluss eines mehrseitigen Formulars vom Framework übernommen. Bei allen Formularen wird in Spring ein ausgereifter Daten-Bindungs-Ansatz verwendet, der Typüberprüfungen und Konvertierungen vorsieht und mit einem sicheren Ausnahme-Behandlungs-Mechanismus ausgerüstet ist. Spring stellt keine Validatoren für Standardtypen zur Verfügung, diese müssen von der Applikation geliefert werden. Die unvollständige Dokumentation ist ein weiteres Minus für Spring.

9.9.3 Maverick

Maverick ist das einfachste der evaluierten Frameworks, es umfasst nur wenige Klassen, die den MVC-Kontrollfluss steuern. Der wichtigste Punkt, der für dieses Framework spricht ist die niedrige Lernkurve. Außerdem basiert Maverick auf einer Architektur, bei der auf die Verwendung von Interfaces an Stelle konkreter Klassen Wert gelegt wird. Dieses minimalistische Framework ist dadurch einfach konfigurierbar und erweiterbar. Von Nachteil sind natürlich die fehlende Validierungs-Unterstützung und das wenig ausgereifte Daten-Bindungs-Konzept des Jarcata *BeanUtil*-Packages.

9.9.4 WebWork2

WebWork2 vereint die positiven Eigenschaften von Struts und Spring. Es bietet genau wie Struts die Möglichkeit Validierung deklarativ zu halten, unterscheidet sich jedoch dadurch, dass auch applikationsspezifische Validatoren verwendet werden können. Außerdem bietet es wie Spring und Tapestry einen ausgereiften Daten-Bindungs-Mechanismus mit Typüberprüfung, Konvertierung und Ausnahmebehandlung. Umgesetzt wird dieser wie bei Tapestry durch den Einsatz der *OGNL*. Zusätzlich ist mit dem Übergang auf Version 2.0 dieses Frameworks die Web-Unabhängigkeit vieler Komponenten durch den Einsatz von Interceptoren erreicht worden. Das Fehlen eines eigenen Formular-Kontrollfluss sowie einer Wizard-Formular-Unterstützung bilden die Nachteile von WebWork2.

9.9.5 Tapestry

Der wesentlichste Vorteil dieses Frameworks ist sein ereignisorientierter Ansatz, von dem vor allem dann profitiert werden kann, wenn im Entwicklerteam starke Kenntnisse der Standard-GUI-Programmierung vorhanden sind. Es kapselt alle webspezifischen Konstrukte in seinen Klassen und zeichnet sich ebenfalls durch einen guten Daten-Bindungs-Ansatz, der genau wie der von WebWork2 oder Spring Typüberprüfungen und Konvertierungen durchführt und die dabei möglicherweise auftretenden Ausnahmen behandelt. Ein weiterer Punkt der für Tapestry spricht, ist die Möglichkeit der deklarativen Validierung, die auch gegebenen Falles erweitert werden kann. Negativ an Tapestry ist eindeutig der hohe Grad an Einarbeitungsarbeit, der notwendig ist um das Framework erfolgreich einsetzen zu können. Weiters ist zu bedenken, dass die Verwendung anderer View-Technologien, als die von Tapestry verwendete Template-Sprache, nicht möglich ist.

9.10 Fazit

Da in dieser Arbeit lediglich der Prototyp der mm:widok implementiert wird und das System voraussichtlich von dem Team der Kunstuniversität weiterentwickelt wird, schien es nicht sinnvoll ein so umfassendes Framework wie Tapestry einzusetzen, da für die Einarbeitung in dieses Framework ein großer Zeitaufwand erforderlich ist.

Obwohl die Custom-Tags von Struts direkt in der Entwicklungsumgebung Dreamweaver von den Designern verwendet werden könnten, ist dieses Framework auf Grund seiner Forderung nach Framework-Abhängigkeit und des unzureichenden Daten-Bindungs-Mechanismus ausgeschieden.

Da Maverick weder Validierung noch Internationalisierung unterstützt und zusätzlich auch keinen ausgereiften Daten-Bindungs-Ansatz vorweisen kann, wurde auch dieses Framework nicht zur Implementierung herangezogen.

Somit kamen WebWork2 und Spring in die engere Auswahl. Beide überzeugen durch die Umsetzung moderner Konzepte wie „Inversion of Control“ und „Aspect Oriented Programming“. Auch im Bereich der Daten-Bindung können beide Frameworks mit ausgereiften Ansätzen dienen. Durch die Wizard-Formular-Unterstützung und das zusätzliche Framework-Angebot für Transaktions-Verwaltung und JDBC-Abstraktion wurde Spring als Implementierungs-Basis für die mm:widok ausgewählt.

Im nächsten Kapitel wird im Zuge der Funktionalitäts- und Implementierungsbeschreibung deutlich, wie die multimediale Wissensdokumentation der Kunstuniversität Linz mit Spring realisiert wurde.

10. Kapitel

Funktionalität und Implementierung der mm:widok

Dieses Kapitel dient zur Beschreibung der multimedialen Wissensdokumentation der Kunstuniversität Linz mm:widok. Zuerst werden die Anforderungen an das System und dessen Funktionalität vorgestellt und anschließend wird die Implementierung des Prototyps der Wissensdokumentation „Top-Down“ beschrieben, das heißt, es wird zuerst ein grober Überblick über das System gegeben, der anschließend immer weiter verfeinert wird.

10.1 Funktionalität

Im Folgenden wird die vom Prototyp implementierte Funktionalität der multimedialen Wissensdokumentation der Kunstuniversität Linz beschrieben.

10.1.1 Authentifizierung und Autorisierung

Das System bietet An- und Abmeldefunktionalität. Die dabei notwendige Authentifizierung der BenutzerInnen erfolgt über den bereits vorhandenen LDAP Server, damit jede/jeder BenutzerIn sein/ihr Passwort auch für die mm:widok verwenden kann. Es gibt im System folgende Rollen, die ein/eine BenutzerIn übernehmen kann:

- AdministratorIn: zur Verwaltung von Benutzerdaten
- ErfasserIn: zur Erfassung von Forschungsergebnissen
- ExporteurIn: zum Export von Forschungsergebnisse
- KontrolleurIn: zur Kontrolle und Freigabe erfasster Forschungsergebnisse
- allgemeine BenutzerIn: zur Suche nach Forschungsergebnissen (für diese Rolle ist keine Anmeldung erforderlich)

Es wird vom System keine Rollenverwaltung geboten, das heißt es existiert keine Benutzeroberfläche um neue Rollen hinzuzufügen bzw. die Semantik der Rollen zu ändern. Einerseits weil diese Funktionalität von Seiten der Kunstuniversität nicht gefordert wurde und andererseits weil neue Rollen bzw. Änderungen der Rollensemantiken auch in der Implementierung umgesetzt werden müssen.

10.1.2 Benutzer-Verwaltung

Weitere Funktionalitäten werden im Bereich der Verwaltung von BenutzerInnen und deren Berechtigungen im System geboten, die im Folgenden aufgelistet werden. Zu all diesen Funktionen ist nur der/die AdministratorIn berechtigt.

Anlegen eines neuen Benutzers/ einer neuen Benutzerin

Es können neue BenutzerInnen angelegt werden. Dazu wird eine Person aus der Liste aller vorhandenen Universitätsbediensteten ausgewählt. Dieser wird eine der hierarchisch strukturierten Organisationseinheiten der Kunstuniversität inklusive aller zugehöriger Untereinheiten und eine oder mehrere Rollen zugeordnet.

Ändern der Benutzer-Einstellungen

Dazu wird eine Person aus der Liste aller vorhandenen BenutzerInnen ausgewählt, deren Einstellungen dann geändert werden können. Es können sowohl Rollen hinzugefügt als auch entfernt werden. Außerdem ist es möglich, der Person eine andere Organisationseinheit inklusive zugehöriger Untereinheiten zuzuordnen.

Löschen eines Benutzers/ einer Benutzerin

Benutzer-Accounts können auch wieder gelöscht werden. Dazu wird eine Person aus der Liste aller BenutzerInnen ausgewählt und gelöscht.

Das System bietet keine eigene Verwaltung von Universitätsbediensteten. Es existiert somit keine Benutzeroberfläche um Universitätsbedienstete zu erfassen, deren Daten zu ändern oder zu löschen, da die Kunstuniversität Linz schon über eine solche Verwaltung verfügt, und dadurch diese verwendet wird.

10.1.3 Erfassung der Forschungsergebnisse

Die Erfassung von Forschungsergebnissen ist durch einen Wizard realisiert worden, damit die Daten von den Erfassern/Erfasserinnen Schritt für Schritt eingegeben werden können. In Abb. 10-1 und Abb. 10-2 ist dieser Wizard dargestellt:

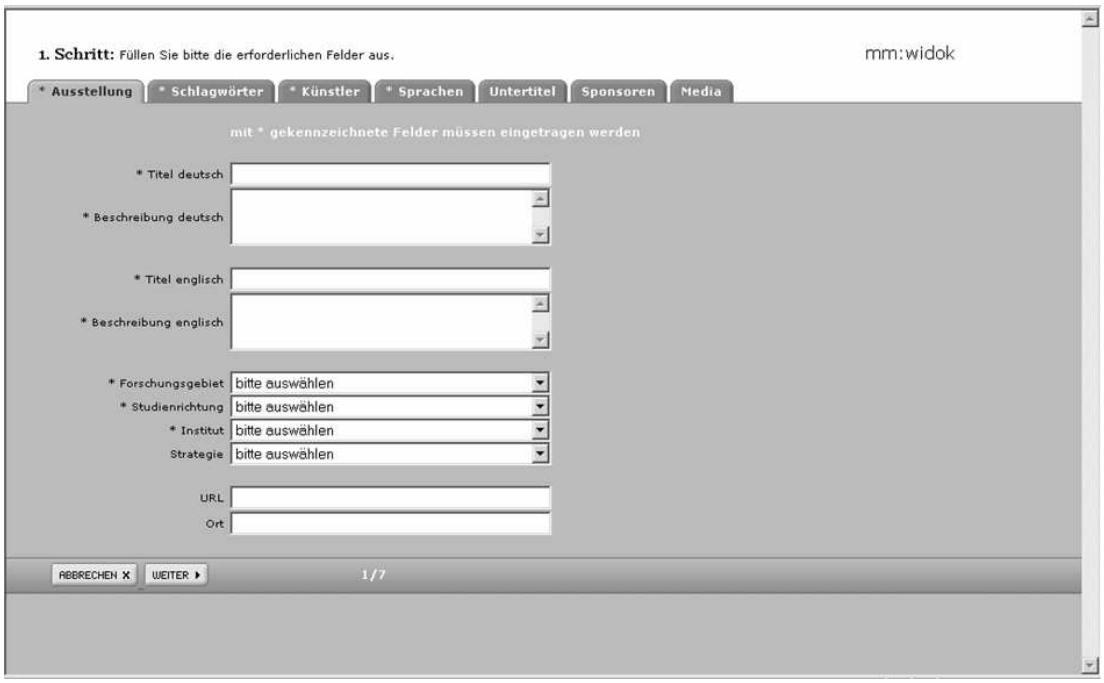

Abb. 10-1: Wizard-Formular zur Erfassung einer Forschungsarbeit

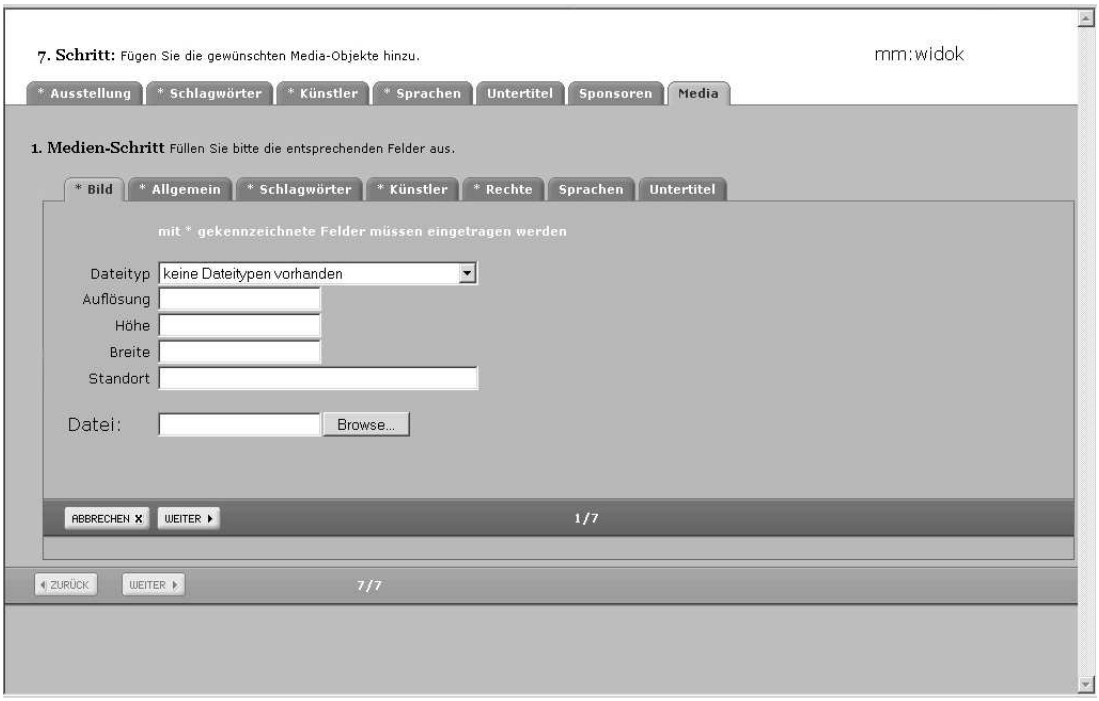

Abb. 10-2: Wizard-Formular zur Erfassung eines Forschungsobjekts

Mit diesem werden zuerst die Metadaten der Forschungsarbeit und dann der Reihe nach die Multimedia-Objekte erfasst. Für die Erfassung der Multimedia-Objekte werden sowohl die Metadaten eingegeben als auch Dateiname und Speicherort für den Upload genannt. Der/die ErfasserIn ist berechtigt alle selbst erfassten Inhalte wieder ändern und löschen zu können.

Da in dieser Diplomarbeit nur ein Prototyp des mm:widok-Systems implementiert wurde, wird nicht für alle Arten von Forschungsarbeiten (Ausstellung, Vortrag, Projekt, Publikation und Veranstaltung) Funktionalität zur Erfassung geboten, sondern nur für die Art *Ausstellung*. Diese wurde ausgewählt, da sich die Kunstuniversität in dieser Art von Forschungsarbeit von anderen Universitäten besonders unterscheidet. Außerdem beschränkt sich die Erfassungsmöglichkeit von Multimedia-Objekten (Bild, Audio, Video und Text) auf *Bilder*. Für jedes erfasste Bild wird ein Thumbnail generiert, der für die Anzeige des Objekts im Suchergebnis verwendet werden kann.

Da Forschungsarbeiten oft in Zusammenarbeit mit externen Personen, Firmen oder Organisationen entstehen, die nicht durch die schon vorhandene, interne Personenverwaltung der Kunstuniversität verwaltet werden können, bietet dieses System die Möglichkeit solche Personen, Firmen oder Organisationen zu erfassen.

Im Folgenden werden weitere Funktionen beschrieben, die in das Design und die Konzeption des Prototyps zwar eingeflossen sind, jedoch für diese Diplomarbeit nicht realisiert wurden.

10.1.4 Freigabe der digitalen Inhalte

Der/die KontrolleurIn hat die Aufgabe das Erfasste im Hinblick auf die Richtigkeit der eingegebenen Informationen zu kontrollieren. Es werden ihm/ihr dazu alle Forschungsarbeiten angezeigt, die noch nicht kontrolliert wurden und einer Organisationseinheiten zugeordnet sind, für die der/die KontrolleurIn autorisiert ist. Daraufhin kann eine der angezeigten Forschungsarbeiten ausgewählt werden, wodurch die erfassten Daten angezeigt werden. Der/die KontrolleurIn kann diese überprüfen. Wenn diese in Ordnung sind, gibt er/sie die Forschungsarbeit frei und macht sie somit öffentlich zugänglich.

10.1.5 Suche nach den digitalen Inhalten

Der/die allgemeine BenutzerIn kann nach digitalen Inhalten suchen, indem er/sie Suchkriterien angibt. Es wurde mit der Kunstuniversität noch nicht erarbeitet, welche Metadaten für die Suche herangezogen werden. Die gefundenen Objekte werden online in geeigneter Form angezeigt.

10.1.6 Export der digitalen Inhalte

Der/die ExporteurIn kann nach bestimmten Forschungsarbeiten suchen. Die Daten der Ergebnisse werden in eine XML-Datei geschrieben, die an einem angegebenen Ort gespeichert wird.

Sowohl die Metadaten der Forschungsarbeiten als auch die der Multimedia-Objekte werden beim Export in ein Dublin-Core-Metadaten-Set DCMS [DCMI03] transformiert. Das DCMS besteht aus folgenden Elementen: Title, Creator, Keywords, Description, Publisher, Contributor, Date, Type, Format, Identifier, Source, Language, Relation, Coverage, Rights. Jedes der Felder ist optional und wiederholbar.

Die gesamte, oben genannte Funktionalität der multimedialen Wissensdokumentation, ist zur besseren Übersicht in Abb. 10-3 in einem UML-Anwendungsfall-Diagramm zusammengefasst:

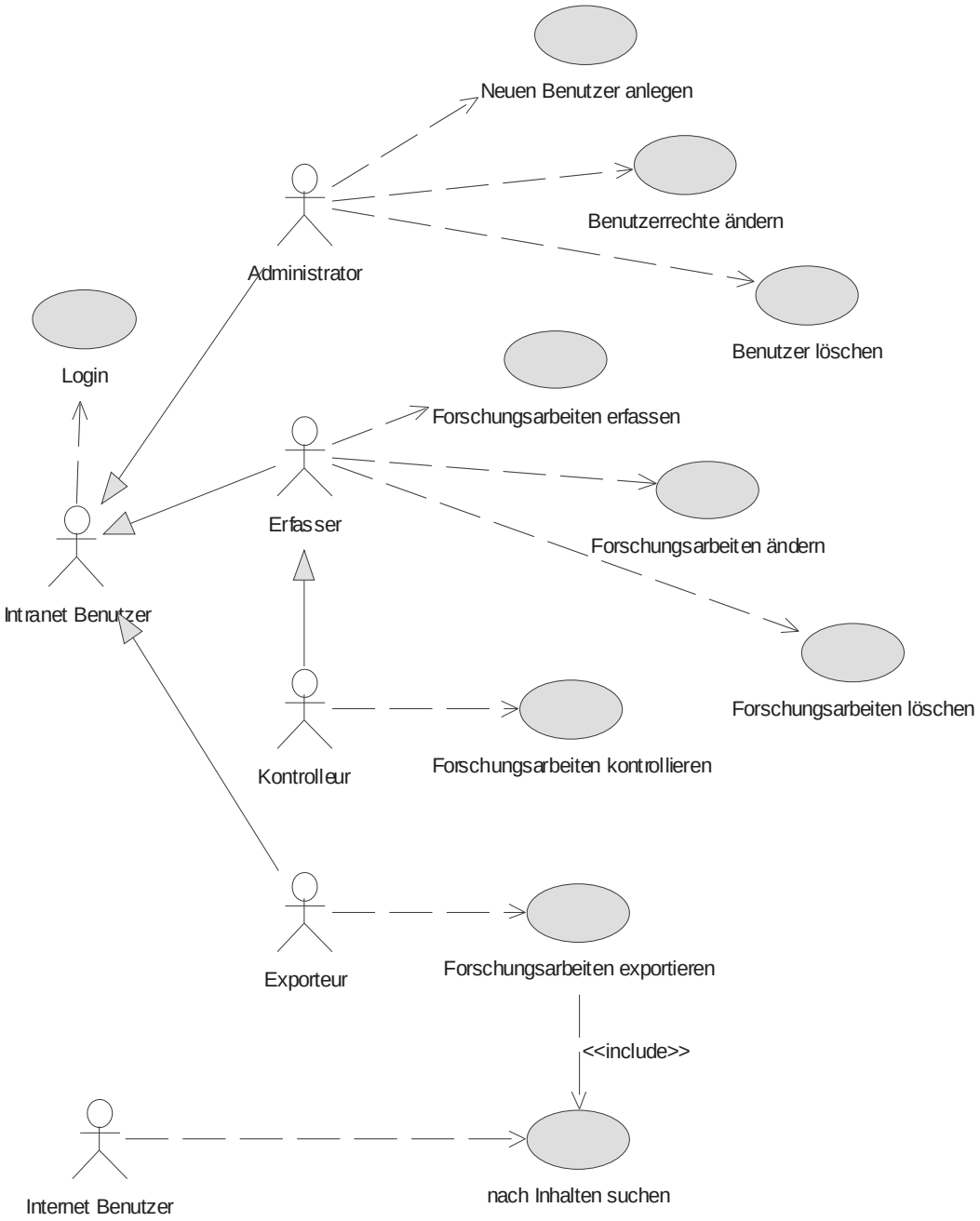

Abb. 10-3: Anwendungsfalldiagramm der mm:widok

10.2 Architektur

Dieser Abschnitt beschreibt die grundlegende Architektur der multimedialen Wissensdokumentation der Kunstuniversität Linz. Abb. 10-4 zeigt zunächst den Zusammenhang zwischen der mm:widok und anderen Systemkomponenten:

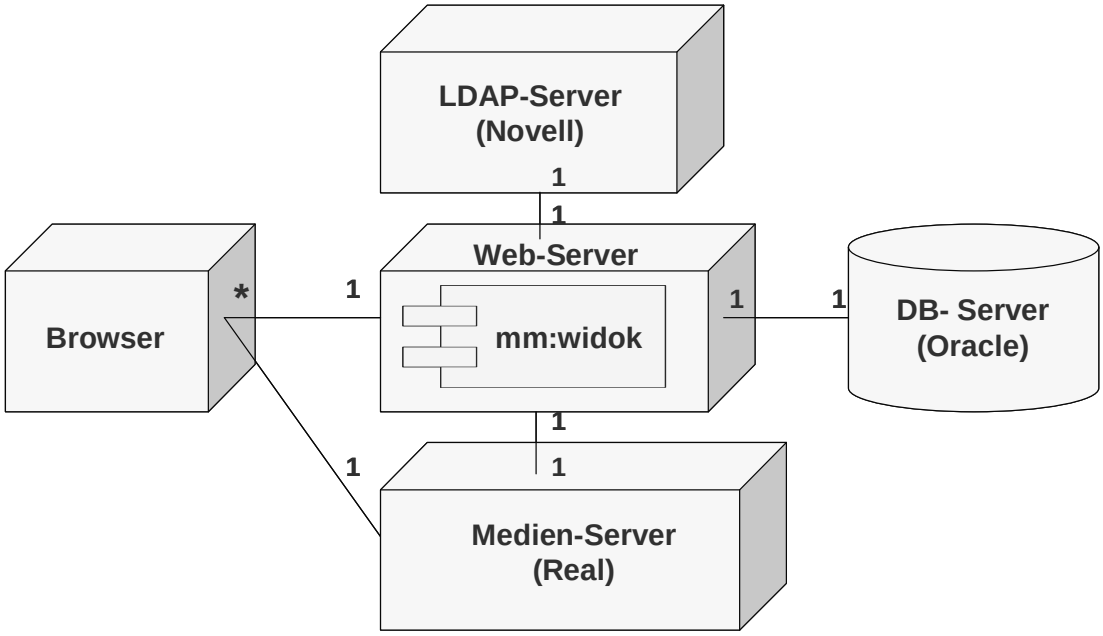

Abb. 10-4: Verteilungsdiagramm der multimedialen Wissensdokumentation

Wie in Abb. 10-4 ersichtlich ist, handelt es sich bei der multimedialen Wissensdokumentation um eine Client-Server-Applikation. Über Browser, die sich am Client befinden, wird das System bedient und die Benutzeroberfläche wird dort dargestellt.

Das System, das auf einem Apache [Apac04] Web-Server mit einer Tomcat 4.2 Servlet-Engine läuft, verwendet einen Novell [Nove04] LDAP-Server, der die Authentifizierung der BenutzerInnen übernimmt. Die Wissensdokumentation speichert die erfassten Metadaten in einer Oracle 9i Datenbank [Orac03] und die Multimedia-Objekte auf einem Medien-Server 8.0 von Real [Real04].

Dem/der BenutzerIn werden die Metadaten des gewünschten Forschungsinhaltes und ein Link auf den Medien-Server, auf dem sich die dazugehörigen Multimedia-Dateien befinden, geliefert. Deshalb hat der Browser nicht nur eine Verbindung zu mm:widok sondern auch zum Medien-Server.

In Abb. 10-5 wird in einem UML-Komponenten-Diagramm eine Ansicht in „Vogelperspektive“ auf das implementierte System dargestellt:

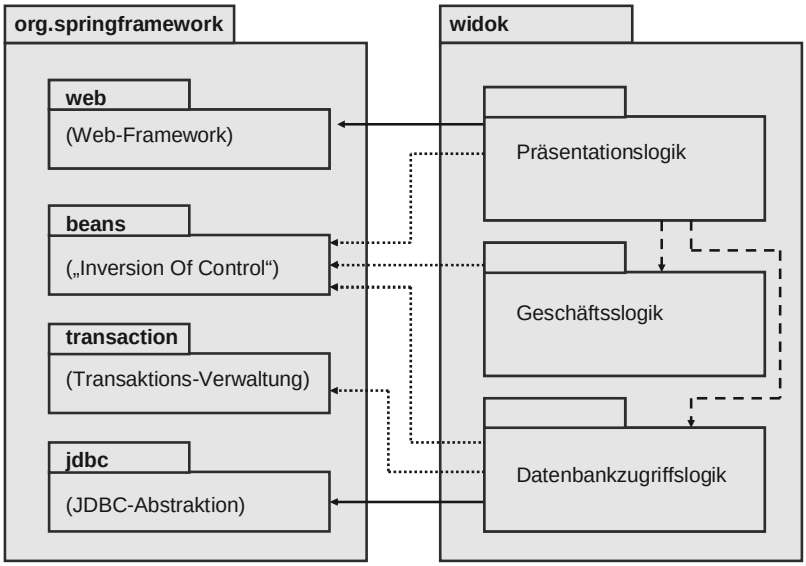

Abb. 10-5: Vogelperspektive auf das System

Das in Abb. 10-5 dargestellte Package *springframework* wird als Rahmen für die multimediale Wissensdokumentation verwendet. Die Klassen des Web-Frameworks werden in dem Package *web* zusammengefasst. Das JDBC-Abstraction-Framework, das den Datenbankzugriff erleichtert, befindet sich im Package *jdbc*. Das Package *beans* wird verwendet, um eine lose Kopplung der Komponenten zu realisieren und dadurch größere Flexibilität zu erreichen. Die gebotene deklarative Transaktions-Verwaltung ist durch die Klassen des Packages *transaction* realisiert.

Das zu implementierende System ist als Package *widok* modelliert und gliedert sich in Präsentations-, Geschäfts- und Datenbankzugriffslogik. Die Präsentationslogik enthält alle webspezifischen Klassen der Applikation und erweitert das Web-Framework. Da die Geschäftsobjekte nicht mehr verändert werden müssen, nachdem sie aus der Datenbank geholt werden, wurden aus Effizienzgründen keine Zugriffsfunktionen in der Geschäftlogik implementiert. Deshalb verwendet die Präsentationslogik direkt die Funktionen der Datenbankzugriffslogik. Die Geschäftslogik ist Web- und Framework unabhängig. Die Datenbankzugriffslogik erweitert das JDBC-Abstraction-Framework.

Die Klassen der beiden Packages *beans* und *transaction* werden von mm:widok nicht explizit verwendet. Das System ist deshalb von diesen unabhängig. Jedoch kann die von diesen Packages zur Verfügung gestellte Funktionalität durch die Spezifikation der mm:widok-Klassen in XML-Dateien verwendet werden. Die punktierte Linie in Abb. 10-5 zeigt diesen Zusammenhang.

10.3 Geschäfts- und Datenbankzugriffslogik

Logisch zusammengehörende Klassen wurden in einer Komponente zusammengefasst, unabhängig davon ob diese Geschäfts- oder Datenbankzugriffslogik implementieren. Abb. 10-6 zeigt diese Komponenten:

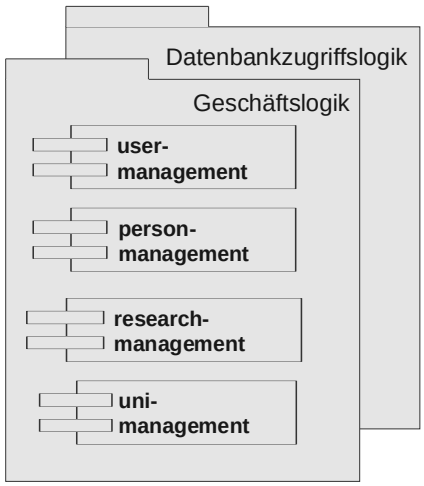

Abb. 10-6: Komponenten der Geschäfts- und Datenbankzugriffslogik

Im Folgenden werden die in Abb. 10-6 dargestellten Komponenten genauer beschrieben.

10.3.1 Benutzer-Verwaltungs-Komponente

Diese Komponente verwaltet die SystembenutzerInnen und deren Berechtigungen, die in Java-Klassen abgebildet werden. Diese Modellierung ist in Abb. 10-7 dargestellt:

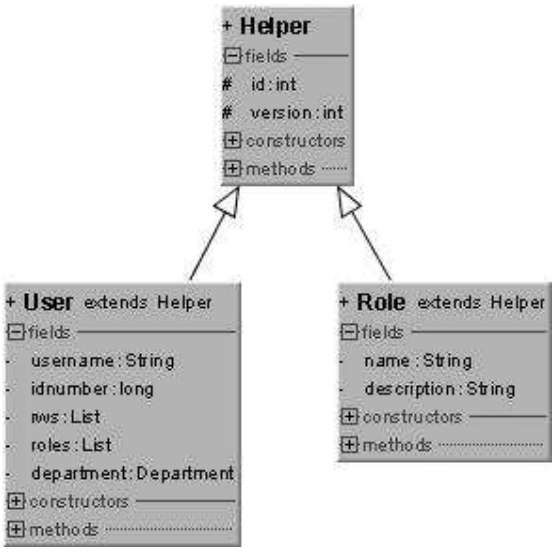

Abb. 10-7: Geschäftsobjekte der Benutzer-Verwaltungs-Komponente

Wie in Abb. 10-7 ersichtlich ist, werden SystembenutzerInnen durch die Klasse *User* repräsentiert. BenutzerInnen können mehrere Rollen besitzen, deshalb besitzt jedes *User*-Objekt ein Attribut *roles* und in dem Feld *department* wird die Organisationseinheit gespeichert, für die der/die BenutzerIn autorisiert ist.

Damit dem/der BenutzerIn zu Beginn ihrer Arbeit die von ihm/ihr erfassten Forschungsarbeiten angezeigt werden können, wird die Liste der Forschungsarbeiten in dem Attribut *rws* gespeichert. Beide Klassen erben von der Klasse *Helper*. Der Grund dafür wird im folgenden Absatz erläutert.

Da es im Mehrbenutzerbetrieb vorkommen kann, dass ein Objekt von mehreren Benutzern gleichzeitig bearbeitet wird, müssen gleichzeitige Zugriffe synchronisiert werden. Dies wird in dieser Arbeit durch ein optimistisches Verfahren umgesetzt. Bei der Implementierung dieses Verfahrens wurde das Design-Pattern *DataMapper* aus [Tura02] benutzt.

Die Grundidee besteht darin, dass bevor Änderungen am Objekt persistent gemacht werden, geprüft wird, ob sich das Objekt seit dem Laden geändert hat. Dazu wird jedes Objekt mit einer Versionsnummer versehen, die nach vollzogener Änderung erhöht wird [Tura02]. Dieses Versionsmanagement wird in einer abstrakten Oberklasse realisiert, da die darin implementierten Methoden für alle persistenten Objekte gleich sind. Da alle Objekte die Attribute *id* und *version* besitzen und diese in der Versionsmangement-Oberklasse erreichbar sein müssen, werden diese Attribute von einer Klasse *Helper* geerbt.

Zur Verwendung der Komponente *usermanagement* werden zwei Interfaces angeboten - *LoginManager* und *UserManager*, die in Abb. 10-8 gezeigt werden:

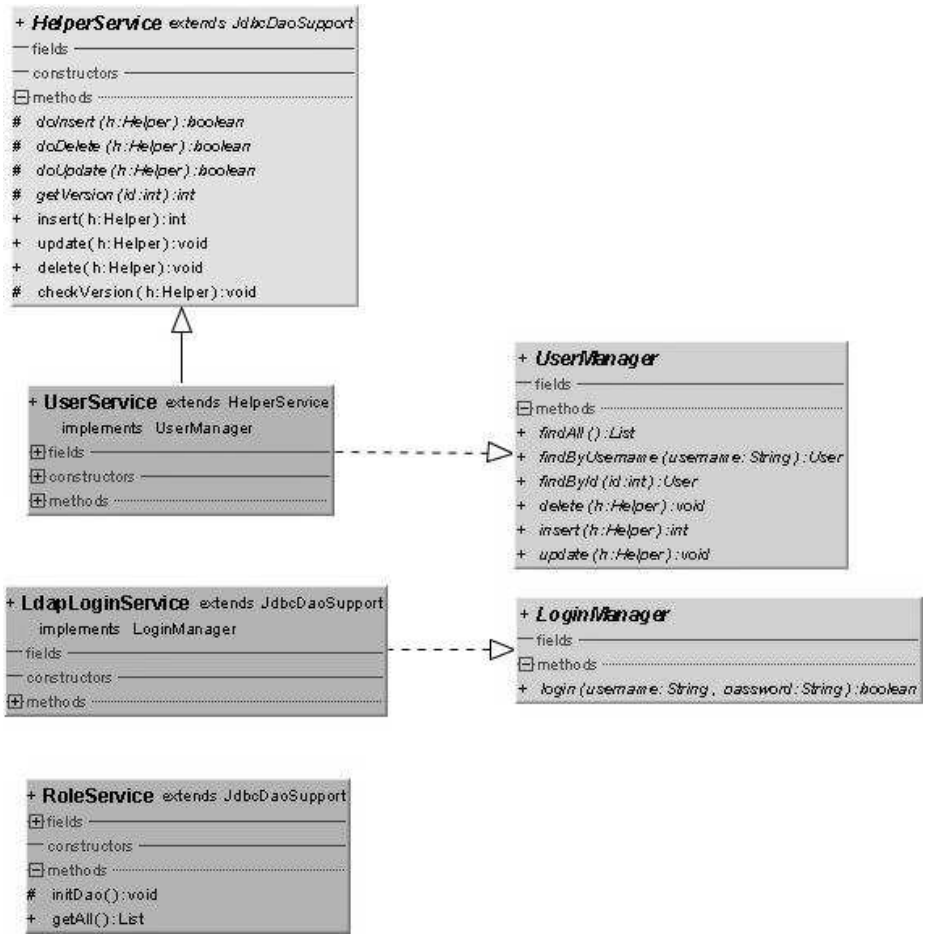

Abb. 10-8: Datenbankzugriff der Benutzer-Verwaltungs-Komponente

Das eine Interface aus Abb. 10-8 bietet nur eine einzige Methode, die dazu dient BenutzerInnen durch Angabe des Benutzernamens und des Passworts im System anzumelden. Ein eigenes Interface für die Benutzeranmeldung ist deshalb sinnvoll, da das System dadurch flexibler wird. Momentan benutzt die Implementierung des *LoginManagers* einen LDAP-Server, um die Benutzerdaten zu autorisieren. Sollte in Zukunft die Login-Information aus einer Datenbank bezogen werden, kann zur Laufzeit eine andere Implementierung geladen werden, die diese Anforderung umsetzt. Das andere Interface bietet Methoden, um Benutzer-Objekte entweder aus der Datenbank zu holen (verschiedene *find*-Methoden) oder in die Datenbank zu speichern (*insert*, *update*, *delete*).

UserService, die Implementierung dieses Interfaces erweitert die Klasse *HelperService*. Diese realisiert die oben erwähnte abstrakte Versionsverwaltungs-Oberklasse. Sie enthält jene Methoden, in denen eine Versionsüberprüfung notwendig ist (*insert*, *update*, *delete*) und Methoden die diese Überprüfung möglich machen

(*getVersion()* und *checkVersion()*). Da es sich bei *HelperService* um eine abstrakte Klasse handelt und ihr weder die Typen der zu speichernden Objekte noch die entsprechenden Datenbank-Tabellen bekannt sind, müssen die Datenbankzugriffe (*doInsert, doUpdate, doDelete, getVerson, checkVersion*) in die konkrete Unterklasse *UserService* ausgelagert werden.

Die beiden Implementierungen *LdapLoginService* und *HelperService* erweitern die Framework-Klasse *JdbcDaoSupport*, da diese Unterstützung für den Umgang mit Datenquellen bietet. Für die Verwaltung der Benutzer-Rollen ist eine eigene Service-Klasse *RoleService* notwendig, die jedoch keines der beiden Interfaces implementiert, da dieses Service nur intern von der Klasse *UserService* verwendet wird. Außerdem fällt auf, dass *RoleService* nicht von der Klasse *HelperService* erbt. Das ist deshalb der Fall, da es für diesen Prototyps nicht vorgesehen ist eine Rollenverwaltung zu realisieren (vergleiche Abschnitt 10.1.1). Somit weist *RoleService* auch keine der Methoden auf, für die Versionsverwaltung notwendig wäre.

Die von außen sichtbaren Methoden *insert()*, *update()* und *delete()* werden in Abb. 10-9 dargestellt:

```
public int insert(Helper h) throws MappingException {
  h.setVersion(1);
  if (!doInsert(h))
    throw new MappingException("insert error: could not
                          insert object into database");
  return h.getId();
}

public void update(Helper h) throws MappingException {
  if (doUpdate(h)) {
    h.incVersion();
  } else {//object is unvalid
    checkVersion(h);
  }
}

public void delete(Helper h) throws MappingException {
  if (h != null && !doDelete(h))
    checkVersion(h);
}
```

Abb. 10-9: insert, update und delete der Klasse UserService

Wie in Abb. 10-9 ersichtlich ist, wird beim Einfügen die Version auf den Standardwert *1* gesetzt. Es kann hier keine Objekt-Kollision entstehen, da es das Objekt vorher noch nicht gegeben hat und es somit kein anderer/ keine andere BenutzerIn es verändern konnte. Die Methode *doUpdate()* führt das Update nur dann aus, wenn die Versionsnummer in der Datenbank noch mit der des Objekts übereinstimmt, falls die

Methode nicht erfolgreich war, wird *checkVersion()* aufgerufen. Der Code diese Methode ist in Abb. 10-10 zu sehen:

```
protected void checkVersion(User u) throws
                                    ConcurrencyException {
  int version = 0;
  try {
     version = getVersion(u.getId());
     if (version != u.getVersion()) {
        throw new ConcurrencyException("User changed …");
     } else {
        throw new RuntimeException("inconsistant db …");
     }
  }catch (Exception e) {
     throw new ConcurrencyException("User doesn't excist");
  }
}
```

Abb. 10-10: checkVersion()-Methode der Klasse UserService

In der *checkVersion()*-Methode aus Abb. 10-10 wird überprüft, ob es sich tatsächlich um einen gleichzeitigen Zugriff oder um einen inkonsistenten Zustand in der Datenbank handelt.

Wie oben schon erwähnt wurde, wird der eigentliche Datenbankzugriff von den Methoden *doGet()*, *doInsert()*, *doDelete()* und *doUpdate()* übernommen. Diese Methoden verwenden dazu vom Framework abgeleitete Klassen, die je ein SQL-„PreparedStatement“ repräsentieren. In Abb. 10-11 und Abb. 10-12 wird die Vorgehensweise beim Datenbankzugriff am Beispiel des Löschens eines Benutzers/ einer Benutzerin veranschaulicht:

```
DeleteUser delUser= new DeleteUser(getDataSource());
[…] //andere Variablen

protected boolean doDelete(User u) throws Exception {
   […] //delete roles
   […] //delete rws
   return delUser.update(u.getId(), u.getVersion())== 1;
}
```

Abb. 10-11: doDelete()-Methode der Klasse UserService

Die Klasse *UserService* besitzt eine private Variable *delUser* vom Typ *DeleteUser*. Diese Variable stellt ein SQL-PreparedStatement dar, kümmert sich jedoch selbständig um den Verbindungsauf- und –abbau, fängt „checked“ Exceptions ab und wandelt sie in „unchecked“ Exceptions um. So wird der Code viel einfacher, lesbarer und wesentlich weniger fehleranfällig, da sich die vielen verschachtelten try/ catch-Blöcke erübrigen. Dieses *DeleteUser*-Objekt kann dann für das Löschen verwendet

werden, indem die von der Framework-Klasse geerbte Methode *update(...)* aufgerufen wird. In Abb. 10-12 ist der Code der *DeleteUser*-Klasse dargestellt:

```
private class DeleteUser extends SqlUpdate {
   public DeleteUser(DataSource ds) {
      super(ds, "delete from widok_user where user_id=?”+
                                          “and version=?");
      declareParameter(new SqlParameter(Types.NUMERIC));
      declareParameter(new SqlParameter(Types.NUMERIC));
      compile();
   }
}
```

Abb. 10-12: DeleteUser-Klasse, die ein SQL-Statement repräsentiert

Die Klasse *DeleteUser* ruft in ihrem Konstruktor, den der Oberklasse auf, und übergibt ihm die Datenquelle *ds* und die SQL-Anweisung, in der für jede Variable ein Fragezeichen eingefügt ist. Für jedes Fragezeichen wird anschließend ein Parameter deklariert. Durch diese übersichtliche Vorgehensweise werden Fehler schneller erkannt und auch einfache Hochkommas stellen kein Problem mehr dar. Sind alle Parameter deklariert worden, muss die Methode *compile()* aufgerufen werden. Ab diesem Zeitpunkt kann das *DeleteUser*-Objekt jederzeit wieder verwendet werden.

In den Methoden *doDelete()*, *doUpdate()* und *doInsert()* wird öfter als einmal auf die Datenbank zugegriffen, da für jeden User auch die Rollen und die erfassten Forschungsarbeiten gelöscht, geändert oder gespeichert werden müssen. Deshalb sollen diese Methoden transaktional sein, damit nicht z.B. ein/eine BenutzerIn ohne die dazugehörigen Rollen eingefügt wird.

Um die Transaktionalität der Methoden zu erreichen, wird die Möglichkeit der deklarativen Transaktions-Verwaltung von Spring genützt. Die dazu notwendige Konfiguration der *applicationContext.xml*-Datei ist in Abb. 10-13 dargestellt:

```
<bean id="transactionManager" class="org.springframework.
          jdbc.datasource.DataSourceTransactionMananger">
   <property name="dataSource">
      <ref bean="dataSource"></ref>
   </property>
</bean>
…
```

```
…
<bean id="myManager" class="org.springframework.
     transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="target">
    <ref bean="userService"></ref>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key= "insert">PROPAGATION_REQUIRED</prop>
      <prop key= "update">PROPAGATION_REQUIRED</prop>
      <prop key= "delete">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Abb. 10-13: Deklarative Transaktions-Verwaltung der Klasse UserService

Wie in Abb. 10-13 ersichtlich ist, muss zuerst die Spring-Klasse *DataSourceTransactionManager* konfiguriert und mit der verwendeten Datenquelle der Applikation in Verbindung gebracht. Anschließend wird die Spring-Klasse *TransactionProxyFactoryBean* deklariert. Dazu werden der zuvor deklarierte *TransactionManager*, die Klasse, deren Methoden transaktional sein sollen (*UserService*), und die Namen dieser Methoden (*insert*, *update*, *delete*) angegeben (vergleiche Abschnitt 5.7.2).

Da das Interface *UserManager* auch verschiedene *find()*-Methoden vorsieht, kann nicht nur die Framework-Klasse *SqlUpdate* verwendet werden, sondern auch die Klasse *MappingSqlQuery*, die von Klassen erweitert wird, die Datenbank-Abfragen umsetzen. Solche Klassen müssen eine Methode *mapRow()* enthalten, die aus einem *ResultSet*, das eine Tabellen-Zeile repräsentiert, das gewünschte Objekt erzeugt. Die geerbte Methode *findObject()* liefert dann als Rückgabewert eine Liste dieser Objekte.

Gibt es mehrere Klassen, die nach den gleichen Objekten suchen, kann eine Klassen-Hierarchie aufgestellt werden, in der die Basis-Klasse die *mapRow()*-Methode enthält und die Unter-Klassen ihre eigene SQL-Anweisung definierten. In den beiden folgenden Abbildungen wird die Anwendung der *mapRow()*-Vererbung am Beispiel des Auffindens eines Benutzers/ einer Benutzerin vorgestellt.

Abb. 10-14 zeigt die Abstrakte Basis-Klasse, in der die Zuordnung zwischen Tabellen-Spalte und Attribut der Klasse *User* definiert wird:

```
private abstract class AbstractUserQuery extends
                                              MappingSqlQuery{
   public AbstractUserQuery(DataSource ds, String sql){
      super(ds, sql);
   }
   protected Object mapRow(ResultSet rs, int i) throws …{
      return new User(rs.getInt("user_id"),
                      rs.getString("username"),
                      rs.getLong("idnumber"),
                      rs.getInt("version"));
   }
}
```

Abb. 10-14: Abstrakte Basis-Klasse für verschiedene Benutzerabfragen

Wie in Abb. 10-14 ersichtlich ist, wird in der *mapRow()*-Methode mit Hilfe des *ResultSet*s ein *User*-Objekt angelegt. Die konkreten Abfrage-Klassen, die die SQL-Anweisungen beinhalten, sind in Abb. 10-15 dargestellt:

```
private class UserQuery extends AbstractUserQuery {
   public UserQuery(DataSource ds) {
      super(ds, "select * from widok_user where user_id=?");
      declareParameter(new SqlParameter(Types.NUMERIC));
      compile();
   }
}
private class UserByNameQuery extends AbstractUserQuery {
   public UserByNameQuery(DataSource ds) {
      super(ds,"select * from widok_user where username=?");
      declareParameter(new SqlParameter(Types.VARCHAR));
      compile();
   }
}
```

Abb. 10-15: Konkrete Klassen, die verschiedene Benutzerabfragen realisieren

Die beiden Klassen *UserQuery* und *UserByNameQuery* erben die *mapRow()*-Methode der Basis-Klasse *AbstractUq*, verwenden jedoch ihre eigenen SQL-Anweisungen, wie in Abb. 10-15 ersichtlich ist. In Abb. 10-16 wird das Datenbank-Schema dargestellt, das sich aus den Klassen der Benutzer-Verwaltungs-Komponente ergibt:

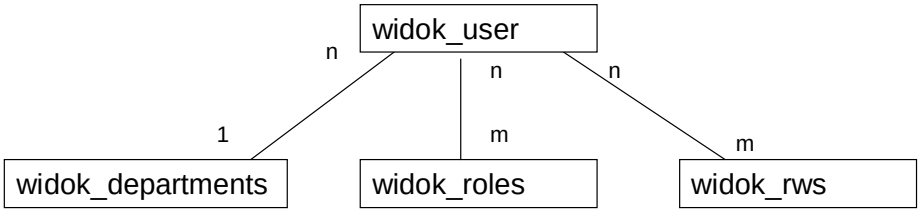

Abb. 10-16: Datenbank-Schema der Benutzer-Verwaltungs-Komponente

Wie in Abb. 10-16 ersichtlich ist, repräsentiert jede Klasse eine Tabelle. Die beiden Listen-Attribute *roles* und *rws* der Klasse *User* sind durch n-zu-m-Verbindungen dargestellt, die in der Datenbank durch die Verbindungstabellen *widok_user_roles* und *widok_user_rws* aufgelöst werden. Die 1-zu-n-Beziehung zwischen *User* und *Department* wird durch die Aufnahme einer Spalte *department_id* als Fremdschlüssel in der *widok_user*-Tabelle realisiert.

10.3.2 Personen-Verwaltungs-Komponente

In dieser Komponente werden die Personen verwaltet. Abb. 10-18 gibt eine Übersicht über die Geschäftsobjekte der Personen-Verwaltung:

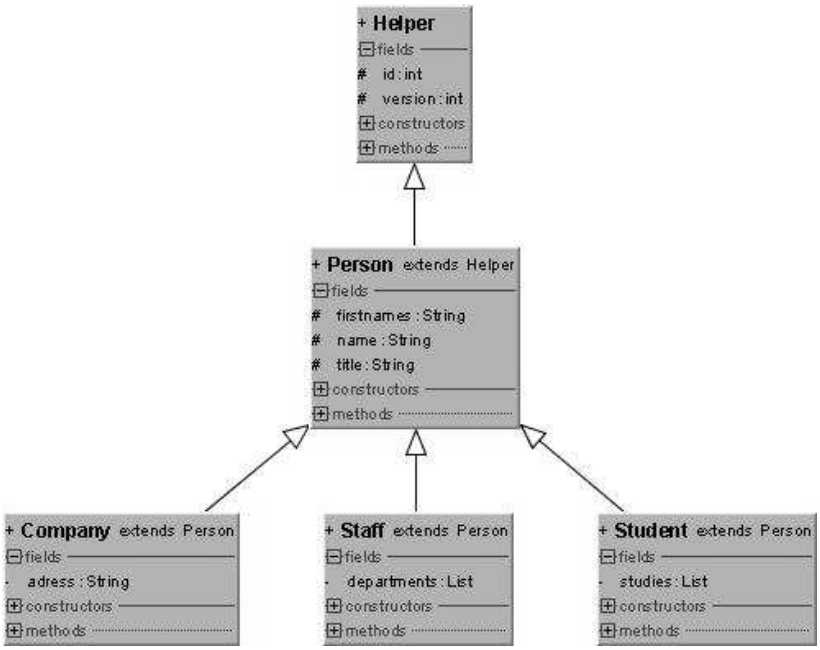

Abb. 10-17: Geschäftsobjekte der Personen-Verwaltungs-Komponente

Jede der verschiedenen Personengruppen (Universitätsbedienstete, Studenten, externe Personen und Firmen) wurde in einer eigenen Klasse abgebildet. Alle Klassen erben wieder aus Gründen des Versionsmanagements von der Klasse *Helper*.

Für jede dieser Klassen existiert ein eigenes Service, das die jeweilige Art von Person verwaltet. Alle Services dieser Komponente implementieren das gleiche Interface, dieser Sachverhalt wird in Abb. 10-18 veranschaulicht:

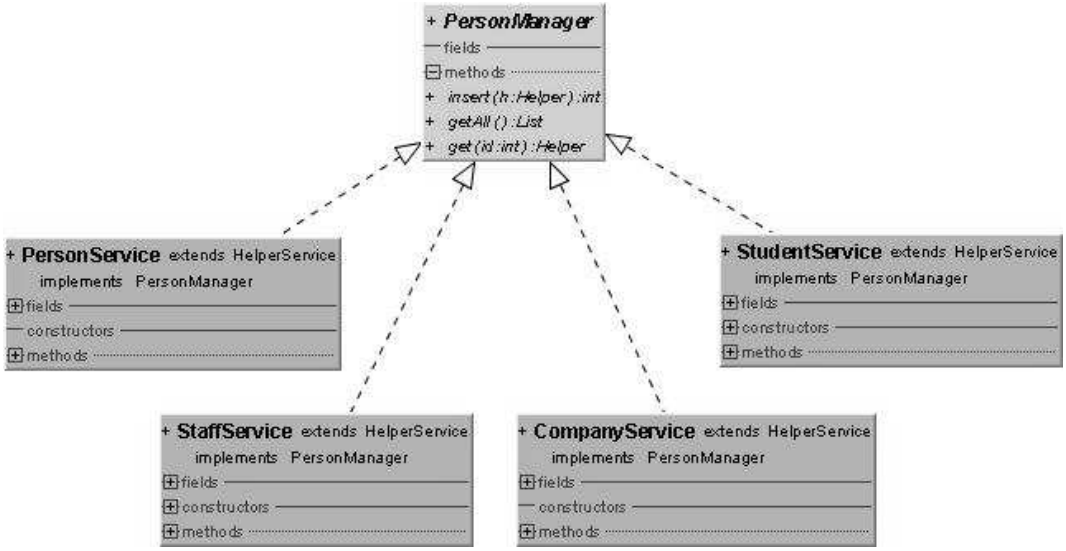

Abb. 10-18: Datenbankzugriff der Personen-Verwaltungs-Komponente

Zur Verwendung der Komponente *personmanagement* wird die Schnittstelle *PersonManager* geboten, die Methoden zum Einfügen und zur Abfrage nach einer oder mehreren Personen enthält. Das Bearbeiten und Löschen ist aus Sicherheitsgründen nicht erlaubt, da es sonst möglich wäre, eine Person versehentlich zu löschen, die noch in einer Forschungsarbeit referenziert wird. Soll dennoch eine Person oder Organisation gelöscht oder geändert werden, muss das auf Tabellenebene geschehen. Bei den Klassen *StaffService* und *StudentService* ist auch die Methode *insert()* nicht erlaubt, da die bestehende Personen-Verwaltung der Kunstuniversität für Studenten und Bedienstete verwendet wird und nur dort Studenten bzw. Bedienstete hinzugefügt, geändert und gelöscht werden können. Somit ist in diesem Fall die Versionsverwaltung momentan eigentlich nicht notwendig, sie ist aber dennoch durch die Erweiterungen der Klasse *HelperService* vorgesehen.

In Abb. 10-19 wird das Datenbank-Schema dargestellt, in dem die Klassen der Personen-Verwaltung gespeichert werden:

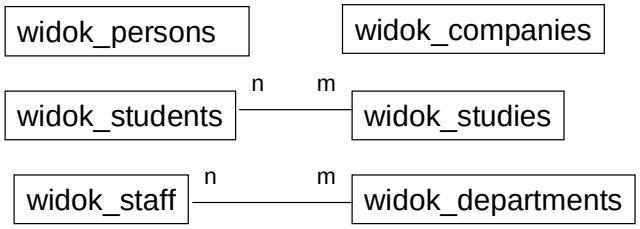

Abb. 10-19: Datenbank-Schema der Personen-Verwaltungs-Komponente

Wie in Abb. 10-19 ersichtlich ist, wurde jeder Personen-Klasse eine Datenbank-Tabelle zugeordnet. Die Listen-Attribute *studies* und *departments* der Klassen Student bzw. Staff werden durch eine n-zu-m-Verbindung dargestellt.

10.3.3 Forschungs-Verwaltungs-Komponente

Die in dieser Komponente verwalteten Forschungsergebnisse werden in Forschungsarbeiten (Ausstellung, Projekt, Publikation, Veranstaltung, Vortrag) und Forschungsobjekte (Audio, Bild, Text, Video) eingeteilt.

Der Zusammenhang zwischen den beiden Kategorien ist, dass Forschungsobjekte Forschungsarbeiten zugeordnet werden können. Zum Beispiel kann es eine Ausstellung (Forschungsarbeit) geben, der mehrere Bilder (Forschungsobjekte) zugeordnet werden. Eine Forschungsarbeit besteht nur aus Metadaten. Ein Forschungsobjekt besteht aus Metadaten und der zugehörigen Multimediadatei.

Attribute, die beiden Typen gemeinsam sind, sind in der Oberklasse *ResearchObject*, die in Abb. 10-20 dargestellt ist, zusammengefasst:

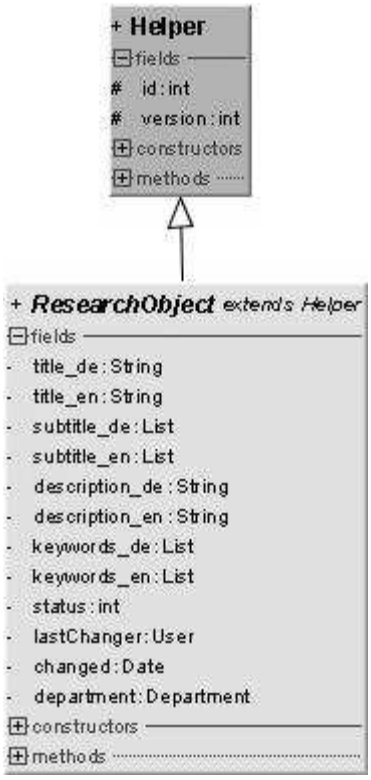

Abb. 10-20: Oberklasse aller Forschungsobjekte und Forschungsarbeiten

Von der in Abb. 10-20 dargestellten Klasse *ResearchObject* erben sowohl alle Forschungsarbeiten als auch alle Forschungsobjekte. Die Oberklasse *Helper* ist wie bei der *User*-Klasse auf Grund der Versionsverwaltung in die Objekt-Hierarchie mit aufgenommen worden. Die verschiedenen Arten von Forschungsarbeiten mit ihren Attributen werden in Abb. 10-21 in einem Klassendiagramm dargestellt:

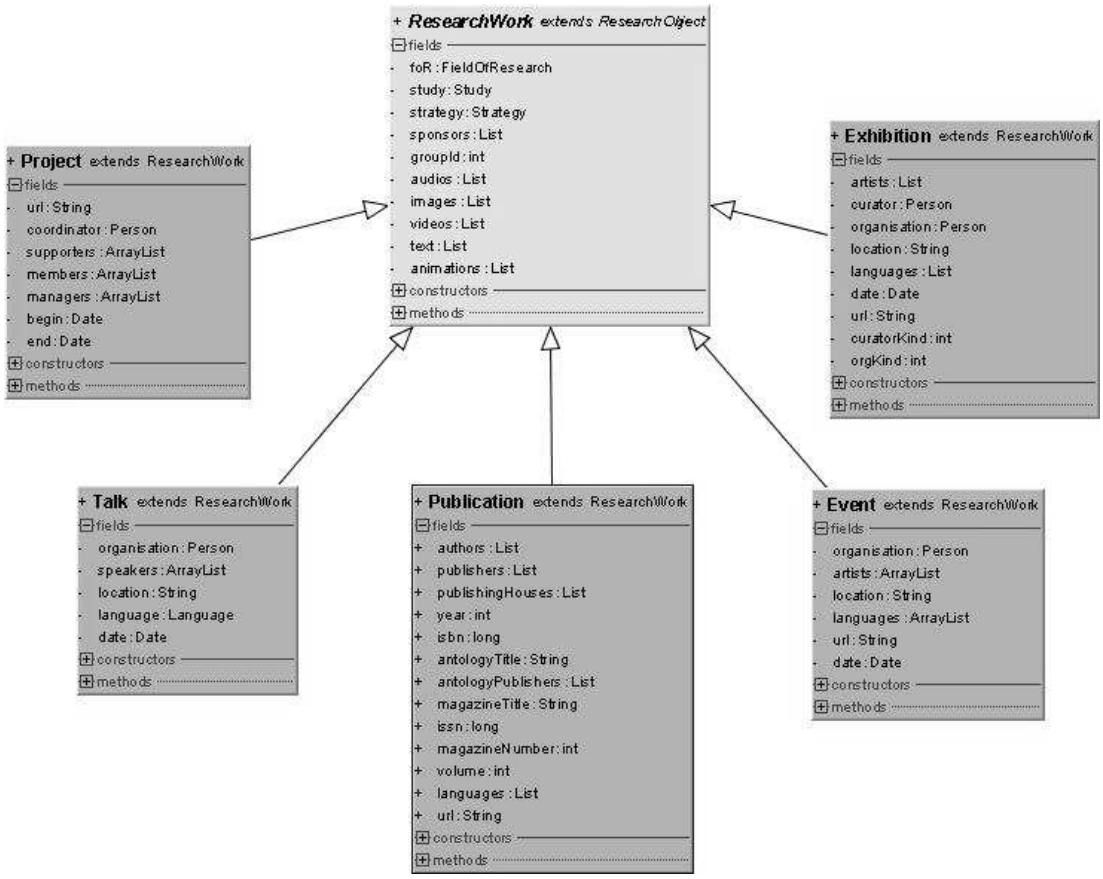

Abb. 10-21: Übersicht über die verschiedenen Forschungsarbeiten

Da in dieser Diplomarbeit nur ein Prototyp des mm:widok-Systems implementiert wurde, wurden nicht für alle Arten von Forschungsarbeiten Verwaltungs-Klassen realisiert, sondern nur für eine Art - die Ausstellung. Diese wurde ausgewählt, da sich die Kunstuniversität besonders in dieser Art von Forschungsarbeit von anderen Universitäten unterscheidet. Da die Attribute *curator* und *organisation* der Klasse vom Typ *Person* sind und diese zur Laufzeit vier verschiedene Ausprägungsmöglichkeiten aufweisen (*Person*, *Company*, *Student*, *Staff*) werden nicht nur die Fremdschlüssel der beiden Personen sondern auch die Personen-Typen in der Datenbank benötigt. Deshalb werden die beiden Attribute *org_kind* und *curator_kind* auch in die Klasse aufgenommen.

In Abb. 10-22 ist das Datenbank-Schema dargestellt, dass sich aus der Klasse *Exhibition* ergibt:

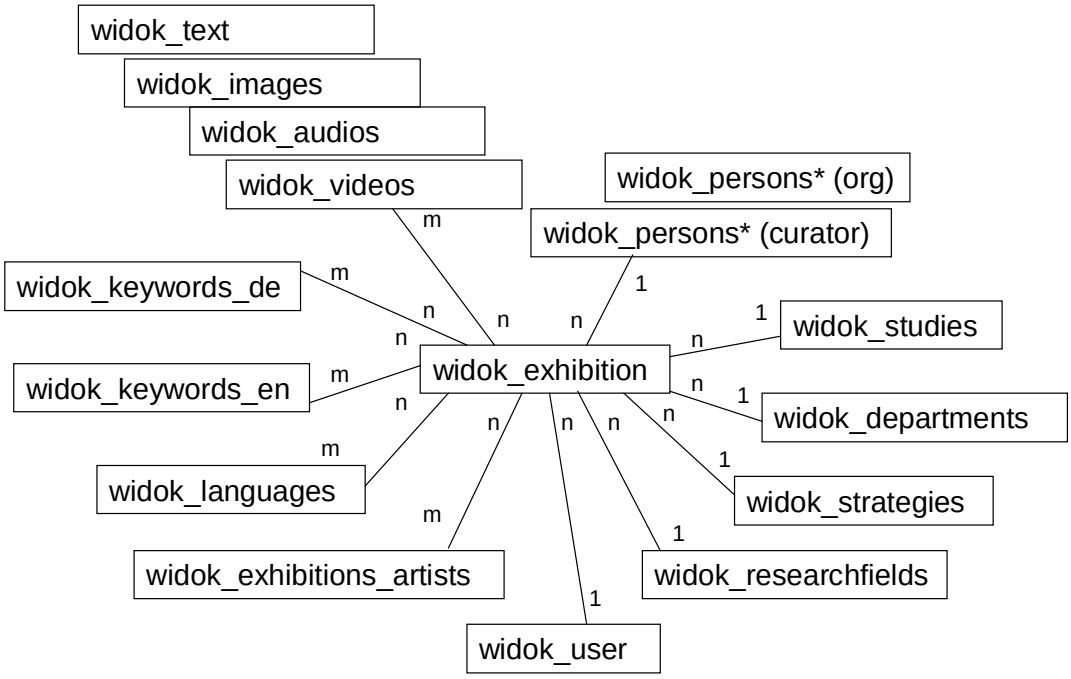

Abb. 10-22: Datenbank-Schema der Klasse Exhibition

In Abb. 10-22 sind die beiden 1-zu-n-Beziehungen, die sich aus den Feldern *curator* und *organisation* ergeben mit einem * eingezeichnet, da die referenzierte Tabelle nicht zwingend *widok_persons* ist, sondern auch *widok_students*, *widok_staff* oder *widok_comapnies* sein kann. In Abb. 10-23 werden die verschiednen Typen der Forschungsobjekte dargestellt:

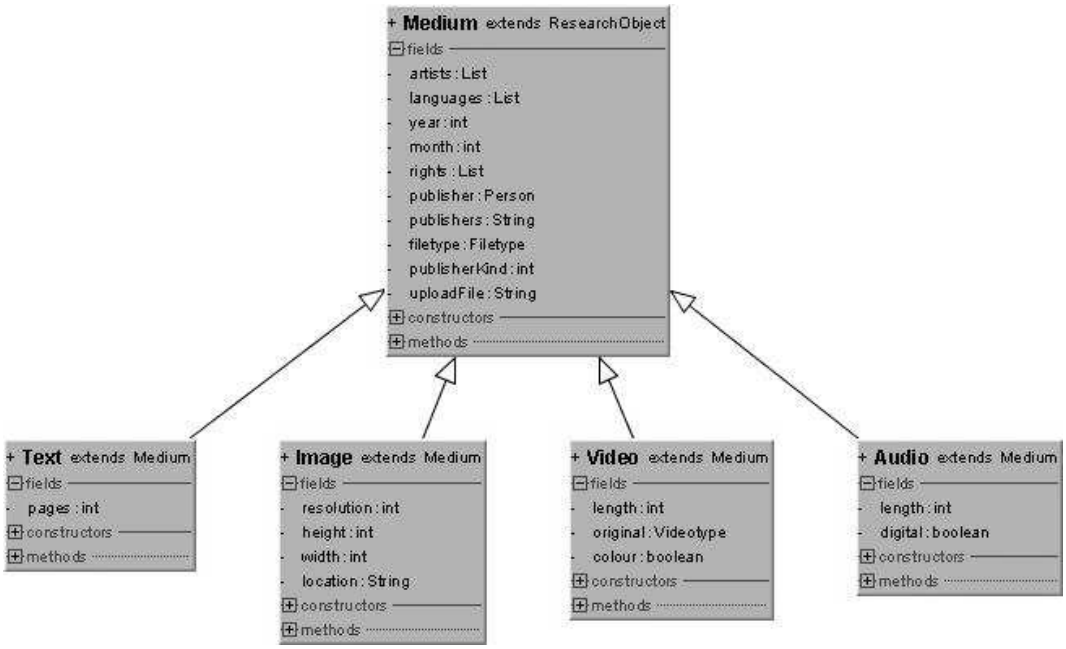

Abb. 10-23: Übersicht über die verschiedenen Typen von Forschungsobjekten

Wie in Abb. 10-23 ersichtlich ist, werden die gemeinsamen Attribute aller Forschungsobjekte in der Oberklasse *Medium* zusammengefasst. Ein wesentliches Attribut dieser Klasse ist *uploadFile*, in dem der Speicherort inklusive Dateinamen des Multimedia-Objekts für den Upload als String gespeichert ist.

Die Zuordnung der Klasse *Image*, die repräsentativ für alle Forschungsobjekte implementiert wurde, auf ein Datenbank-Schema wird in Abb. 10-24 veranschaulicht:

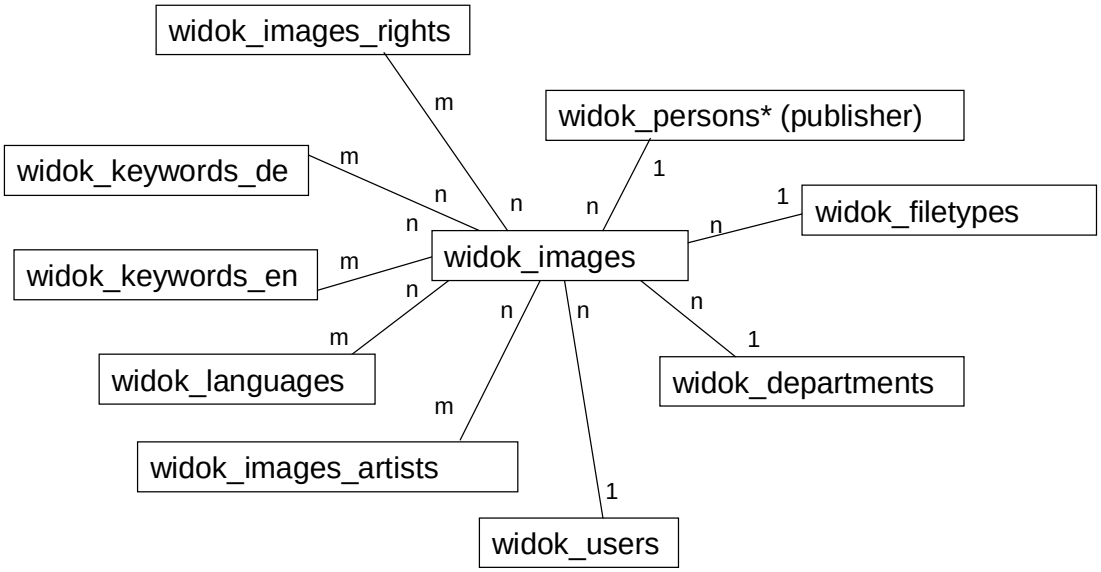

Abb. 10-24: Datenbank-Schema der Klasse Image

Wie bei der Klasse *Exhibition* ist die Tabelle *widok_persons* auf Grund der zur Laufzeit verschiedenen Ausprägungsmöglichkeiten des Attributs *publisher* mit * gekennzeichnet.

Die in Abb. 10-21 und Abb. 10-23 dargestellten Geschäftsobjekte werden durch Service-Klassen verwaltet. Diese kapseln den Datenbankzugriff. In Abb. 10-25 sind die zwei Services abgebildet, die für diesen Prototyp implementiert wurden. Alle Service-Klassen implementieren das *ResearchManager*-Interface, das ebenfalls in Abb. 10-25 gezeigt wird.

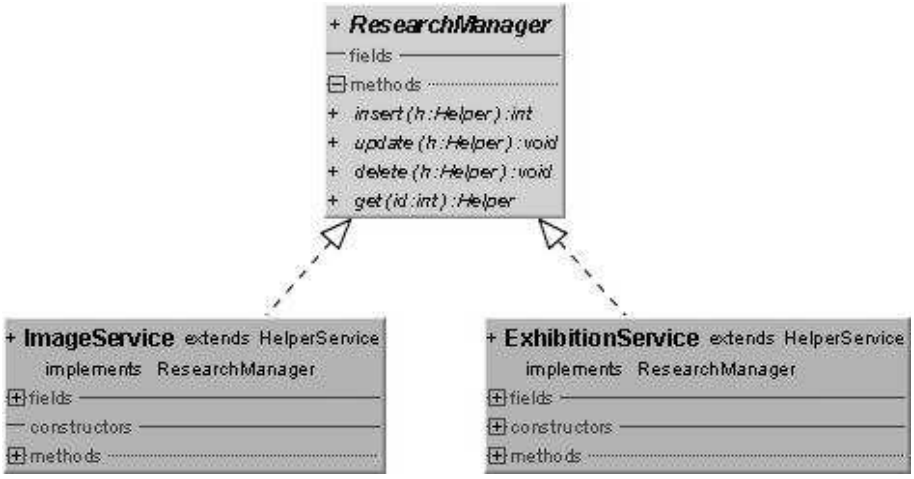

Abb. 10-25: Zwei Service-Klassen der Komponente researchservice

Das zur Verfügung gestellte Interface *ResearchManager* bietet Methoden, um Forschungsergebnisse entweder aus der Datenbank zu holen (*get*) oder in der Datenbank zu manipulieren (*insert*, *update*, *delete*). Diese Methoden sind genau wie jene in den anderen Komponenten mit Hilfe des JDBC-Abstraction-Frameworks implementiert und mittels deklarativer Transaktions-Verwaltung transaktional gestaltet worden (vergleiche Abschnitt 5.7. und 11.3.).

Der Unterschied der Service-Klassen, die Forschungsobjekte verwalten, zu allen anderen besteht in der Notwendigkeit nicht nur Metadaten sondern auch Multimedia-Objekte zu verwalten. Das wird im Folgenden am Beispiel der Klasse *ImageService* deutlich gemacht.

Die *insert()*-Methode des *ImageServices* muss zusätzlich zur Metadatenspeicherung die Bild-Datei nach dem Upload in die Datenbank laden, dort mit Hilfe des Bildbearbeitungs-Moduls „InterMedia“ von Oracle ein „Thumbnail“ generieren und dieses sowie die ursprüngliche Datei auf den Medien-Server speichern und eine Referenz auf die beiden Objekte in der Datenbank behalten.

Dazu wird folgendermaßen vorgegangen: Die Datei liegt bereits am Medien-Server, denn der Controller hat nach dem Abschicken des Formulars die Datei vom Rechner des Benutzers in ein bestimmtes Verzeichnis auf den Web-Server geladen. Da der Medien-Server mit diesem über ein Netzwerk verbunden ist und das Verzeichnis des Web-Servers, in welches die Multimedia-Dateien geladen werden, einen „Mountpoint“ des Medien-Servers darstellt, ist dieses Verzeichnis zwar physisch am Web-Server, logisch gehört es aber zum Medien-Server.

Oracle bietet die externe Verwaltung von Multimedia-Dateien durch das Konzept der *bfile* an. Dabei werden lediglich die Metadaten der Datei sowie der Speicherort in der

Datenbank gespeichert und das eigentliche Multimedia-Objekt kann auf einem File-Server oder wie im Fall der mm:widok auf einem Media-Server liegen. Genau wie bei der internen Verwaltung, bei der die Dateien auch physisch in der Datenbank gespeichert werden, ist bei der Typdefinition der Tabellenspalten, die Multimedia-Objekte enthalten sollen, die Verwendung der Oracle InterMedia Datentypen notwendig. Abb. 10-26 zeigt einen Ausschnitt des Erstellungsskripts für die Tabelle *widok_images*:

```
create table widok_images(
  /* normale Attribute */
  image_id   number,
  version   number,
  title_de   varchar2 (1000),
  title_en   varchar2 (1000),
  […]
  /* Daten */
  image     OrdSys.OrdImage,
  thumbnail  OrdSys.OrdImage,
);
```

Abb. 10-26: Ausschnitt aus dem Skript zur Erstellung der Tabelle widok_images

Wie aus Abb. 10-26 hervorgeht, sind diese Datentypen (*OrdImage, OrdAudio, OrdVideo* und *OrdDoc*) in dem Package *OrdSys* zusammengefasst. Um eine Bild-Datei in der Datenbank in ein *OrdImage*-Objekt zu speichern, kann das JDBC-Abstraction-Framework von Spring nicht verwendet werden. Da zur Speicherung eines *OrdImages* die Funktion *getCustomDatum()* der Klasse *OracleResultSet* notwendig ist und das JDBC-Abstraction-Framework jedoch intern mit der Klasse *ResultSet* arbeitet. Es müsste daher direkt mit der JDBC-API gearbeitet werden, damit die Klasse *OracleResultSet* verwendet werden kann.

Deshalb wurde die Funktionalität des Initialisierens, der Thumbnail-Generierung und des Speicherns in eine Pl*SQL-StoredProcedure namens *insert_image* ausgelagert. Diese Entscheidung wurde auch deshalb getroffen, da das System ohnehin nicht Datenbank-unabhängig ist, weil die Oracle-Datentypen zur Speicherung verwendet werden.

In Abb. 10-27 wird die StoredProcedure *insert_image* dargestellt:

```
procedure insert_image(filename in varchar,
                       img_id out number)
is
    img OrdSys.OrdImage;
    tn OrdSys.OrdImage;
    ctx raw(4000):=null;
begin
     -- get new id from sequence
     select image_seq.NEXTVAL into img_id from dual;

     -- initialise
     insert into widok_images(image_id,image,thumbnail)
     values (
       img_id,
       OrdSys.OrdImage.init('file','ORDIMGDIR',filename),
       OrdSys.OrdImage.init()
     );

     -- locking the empty images
     select image, thumbnail into img,tn from widok_images
     where image_id= img_id for update;

     --set properties and generate thumbnail
     img.import(ctx);
     img.setProperties();
     img.processCopy('maxscale= 32 32, fileformat= GIFF',
                     tn);
     tn.setProperties();

     --update the new images
     update widok_images set image= img, thumbnail= tn
     where image_id=img_id;
end;
```

Abb. 10-27: StoredProcedure zur Speicherung der Bild-Datei

In der StoredProcedure aus Abb. 10-27 wird zuerst eine neue *id* aus der Sequence *images_seq* geholt, diese wird später von der Prozedur zurückgegeben. Anschließend müssen die Datenbankfelder, die vom Typ *OrdImage* sind mit der *init()*-Methode der Klasse *OrdImage* initialisiert werden. Die dabei verwendeten Parameter haben folgende Bedeutung: *'file'* bedeutet, dass sich die Bild-Datei auf einem lokalen Dateisystem befindet, *'ORDIMGDIR'* ist der Name des Verzeichnisses, in welches die Dateien beim Upload geladen werden. Dieses Verzeichnis muss mit dem Befehl *create or replace directory ORDIMGDIR as '<directory>'* erstellt worden sein und muss der *init()*-Methode in Großbuchstaben übergeben werden. Der dritte Parameter enthält den Dateinamen. Nach der Initialisierung muss die Zeile mit einem *SELECT*

FOR UPDATE gesperrt werden. Dadurch kann die *setProperties()*-Methode, die Metadaten wie Dateigröße, Bildbreite, Mimetype, etc. eruiert, und auch die *processCopy()*-Methode ausgeführt werden. Diese Methode bewirkt, dass ein Thumbnail der Größe 32 x 32 Pixel im GIF-Format generiert und in der Variable *tn* gespeichert wird. Abschließend werden sowohl das Original-Bild als auch das Thumbnail wieder aktualisiert.

Nicht nur Abfragen und Update-Statements können mit Hilfe des JDBC-Abstraction-Frameworks umgesetzt werden, sondern auch der Aufruf von StoredProcedures. In Abb. 10-28 wird die Klasse *InitAndInsertImage* dargestellt, die den Aufruf der StoredProcedure aus Abb. 10-27 realisiert:

```
private class InitAndInsertImage extends StoredProcedure{
  public InitAndInsertImage(DataSource dataSource) {
    super(dataSource, "insert_image");
    declareParameter(new SqlParameter(Types.VARCHAR));
    declareParameter(new SqlOutParameter(Types.NUMERIC));
    compile();
  }
  public int execute(String filename){
    Number id= new Integer(0);
    Map in= new HashMap();
    in.put("filename",filename);
    in.put("img_id", id);
    Map out= execute(in);
    id= (Number)out.get("img_id");
    return id.intValue();
  }
}
```

Abb. 10-28: InitAndInsertImage-Klasse

Die Klasse *InitAndInsertImage* aus Abb. 10-28 erweitert die Framework-Klasse *StoredProcedure*. Genau wie bei einer *MappingSqlQuery* oder der *SqlUpdate*-Unterklasse wird die SQL-Anweisung – in diesem Fall der Aufruf der StoredProcedure *insert_image* – gemeinsam mit der Datenquelle dem Konstruktor der Oberklasse übergeben. Anschließend werden Parameter deklariert und die *compile()*-Methode wird aufgerufen. Die Ein- und Ausgabe-Parameter einer StoredProcedure werden mit Hilfe einer *Map* übergeben. Abb. 10-29 zeigt die Verwendung dieser Klasse.

```
public class ImageService extends HelperService implements
                    ResearchManager{
  private InitAndInsertImage initSP;
  […] //andere Variablen

  protected void initDao() throws Exception {
    […] // Anlegen der anderen Klassen
    initSP= new InitAndInsertImage(getDataSource());
  }

  protected boolean doInsert(Helper h) throws
                                MappingException {
    […]  //filename aus variable uploadFile
    img.setId(initSP.execute(filename));
    […] //einfügen der Metadaten
  }
}
```

Abb. 10-29: doInsert()-Methode der Klasse ImageService

Wie in Abb. 10-29 ersichtlich ist, hat die Klasse *ImageService* eine Variable *initSP* vom Typ *InitAndInsertImage*, die in der *initDao()*-Methode, die durch die Oberklasse *JdbcDaoSupport* zur Verfügung gestellt wird initialisiert wird. Durch den Aufruf der Methode *execute()* wird die StoredProcedure *insert_image* aus Abb. 10-27 ausgeführt und liefert die neue *id* für das *Image*-Objekt zurück.

10.3.4 Universitäts-Verwaltungs-Komponente

In dieser Komponente werden die Studienrichtungen, die Abteilungen, die Forschungsgebiete und die Forschungsstrategien verwaltet. In Abb. 10-30 sind die Klassen aus dem Package *beans* der Komponente *unimanagement* abgebildet:

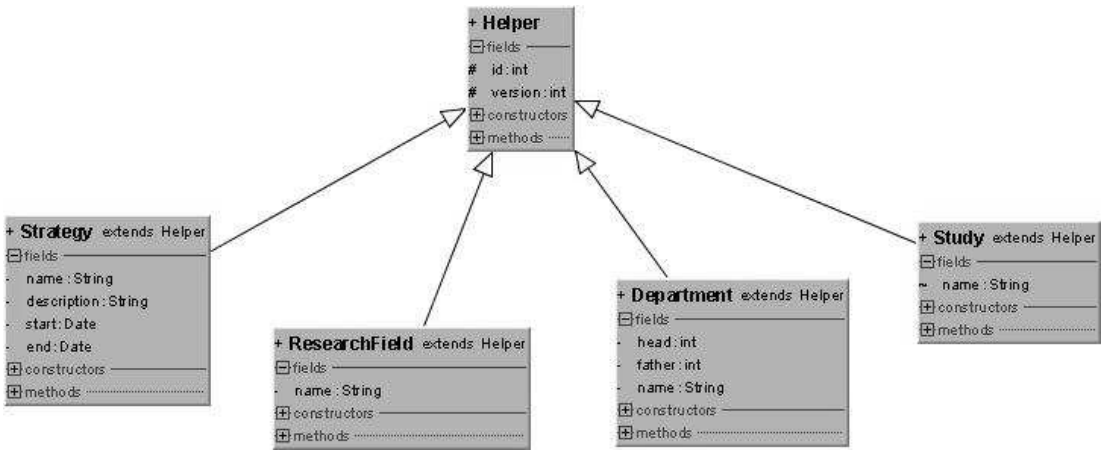

Abb. 10-30: Geschäftsobjekte der Komponente unimanagement

Jede der Klassen aus Abb. 10-30 ist in der Datenbank durch eine eigene Tabelle vertreten. Wie aus der Abbildung ersichtlich ist, erben auch alle Klassen dieser Komponente von der Klasse *Helper*. Da die gezeigten Objekte in der momentanen Implementierungs-Phase nur abgefragt und nicht eingefügt, geändert oder gelöscht werden können, dient diese gemeinsame Oberklasse im Moment nur der Ermöglichung einer einheitlichen Schnittstelle, die in Abb. 10-31 dargestellt wird:

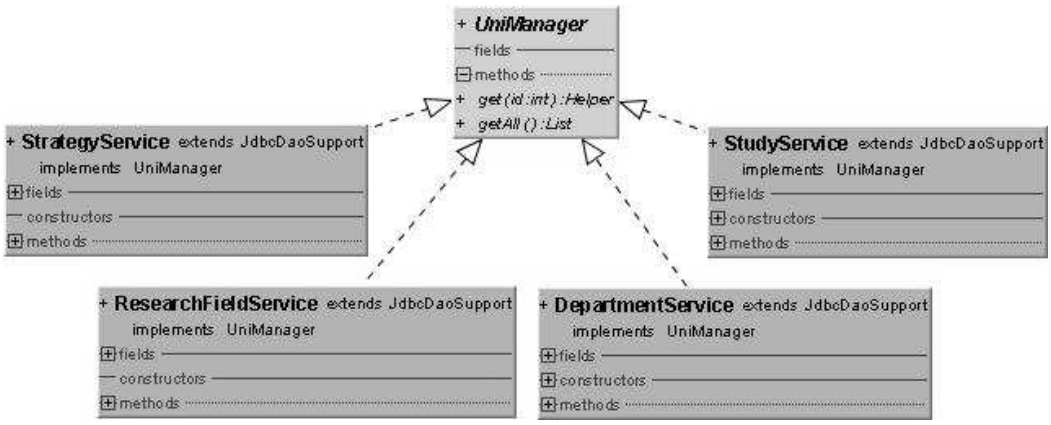

Abb. 10-31: Datenbankzugriff der Komponente unimanagement

Wie Abb. 10-31 zeigt, stellt die Komponente *unimanagement* ein Interface *UniManager* zur Verfügung, um die oben vorgestellten Geschäftsobjekte aus der Datenbank zu holen.

10.4 Präsentationslogik

Um eine saubere Schichtentrennung zu realisieren, werden in der Präsentationslogik alle Klassen zusammengefasst, die für die Darstellung der Web-Applikation im Browser zuständig sind. Nachdem die multimediale Wissensdokumentation nach dem „Model 2“-Entwurfsmuster (vergleiche Abschnitt 2.1) aufgebaut ist und das Spring Web-Framework verwendet, übernehmen die Klassen der Präsentationslogik die Funktion der Controller (vergleiche Abschnitt 3.1). Abb. 10-32 zeigt, wie die Präsentationslogik gegliedert ist:

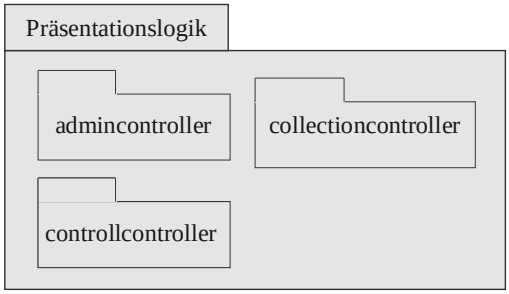

Abb. 10-32: Interne Struktur der Präsentationslogik

Wie in Abb. 10-32 ersichtlich ist, gibt es für jede der Benutzerrollen “Administration”, “Kontrolle” und “Erfassung” ein eigenes Package. In diesen Packages sind alle Controller zusammengefasst, die für die Präsentation der Webseiten für den jeweiligen Benutzer/ die jeweilige Benutzerin verantwortlich sind. Für den Export und die Suche von Forschungsergebnissen werden keine Controller benötigt, da die Präsentation der beiden Anwendungsfälle in Flash [Macr04] mit Hilfe von ActionScript [Macro04] realisiert werden soll. Die Umsetzung dieser Anwendungsfälle liegt außerhalb dieses Diplomarbeitsprojekts.

Alle Controller implementieren das *Controller* Interface des Spring-Web-Frameworks, entweder direkt in dem sie dessen Methode

```
ModelAndView handleRequest (request, response)
```

implementieren, oder indem sie eine der zur Verfügung gestellten Controller-Super-Klassen (wie z.B. *SimpleFormController* oder *AbstractWizardFormController*) ableiten (vergleiche 5.1.2). In den folgenden Abschnitten wird die Arbeitsweise der Controller der einzelnen Benutzerrollen beschrieben.

10.4.1 Administration

Nachdem der/die AdministratorIn sich am System angemeldet hat, bekommt er/sie eine Übersicht über alle vorhandenen BenutzerInnen im System. Für diese Übersicht ist der *UserManagementController* zuständig. Abb. 10-33 zeigt den Code dieses Controllers:

```
public class UserManagementController implements Controller
{
    private UserService userService;
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    public ModelAndView handleRequest(
       HttpServletRequest reqest,
       HttpServletResponse response) throws Exception {
        List userlist = userService.findAll();
        Map model = new HashMap();
        model.put(“userlist”, userlist);
        return new ModelAndView(“usermanager”,model);
    }
}
```

Abb. 10-33: UserManagementController

Wie Abb. 10-33 zeigt, ist dieser Controller seht einfach. Es handelt sich um kein Formular und seine einzige Aufgabe ist, der Web-Seite *usermanager.jsp* die Benutzerdaten zur Verfügung zu stellen. Diese werden mittels dem *UserService* (vergleiche Abschnitt 10.3.1) aus der Datenbank geholt und in eine *HashMap*, die als Model fungiert, unter dem Namen *„userlist“* einfügt. Über diesen Namen kann die JSP-Seite auf die Liste der Benutzerdaten zugreifen und diese darstellen.

Am Code des *UserManagementControllers* fällt auf, dass die Variable *userService* zwar verwendet wird, jedoch nicht instanziert wird. Diese Aufgabe übernimmt das Geschäftslogik-Framework von Spring, es ist dafür nur die Änderungsmethode *setUserService(...)* notwendig. So kann zur Laufzeit bestimmt werden, welche Implementierung des *UserServices* für diese Aufgabe verwendet werden soll. Auch bei allen anderen Controllern wird diese Möglichkeit der losen Kopplung verwendet.

Ausgehend von der *usermanager.jsp* Seite hat der/die AdministratorIn die Möglichkeit entweder neue BenutzerInnen anzulegen oder bestehende BenutzerInnen zu ändern bzw. zu löschen. Für alle diese Möglichkeiten existiert ein eigener Controller. An diesen wird der Request weitergegeben, wenn der/ die AdministratorIn eine dieser Möglichkeiten auswählt. In Abb. 10-34 sind alle Controller dieses Packages dargestellt:

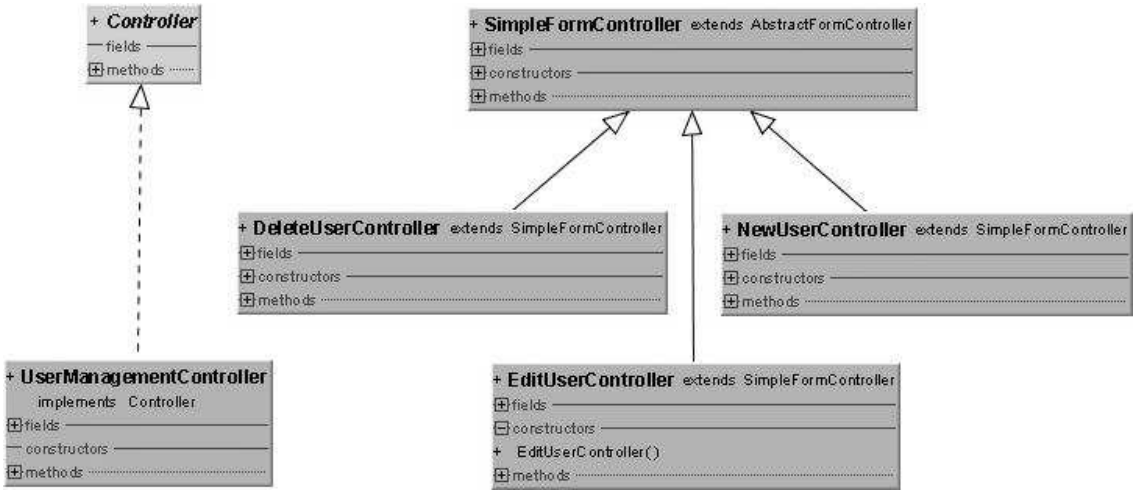

Abb. 10-34: Klassendiagramm der Administrator-Controller

Wie in Abb. 10-34 zu sehen ist, erweitern die Controller, die für das Anlegen, Löschen und Ändern von BenutzerInnen zuständig sind, die Framework-Klasse *SimpleFormController*. Im Gegensatz zu der Klasse *UserManagementController*, stellen die diesen Controllern zugeordneten JSP-Seiten Formulare dar, denn die Interaktion mit dem/der AdministratorIn ist für diese Vorgänge notwendig.

Die Klasse *SimpleFormController* vereinfacht die Implementierung eines Formular-Controllers. Ohne Framework würde man zwei Controller-Klassen implementieren, eine für das Anzeigen des leeren Formulars und eine für die Verarbeitung. Durch die Verwendung des *SimpleFormControllers* ist nur eine Klasse notwendig, in der beide Funktionen zusammengefasst werden. Außerdem bietet das Framework Daten-Bindung. Es kann daher ein *User*-Objekt an das Formular gebunden werden, und so automatisch dessen Attribute mit den von dem/der AdministratorIn eingegebenen Werten befüllt werden (vergleiche 5.1.2).

In Abb. 10-35 wird der Anwendungsfall „Anlegen eines neuen Benutzers/ einer neuen Benutzerin“ in einem Sequenz-Diagramm dargestellt. Dieser Ablauf ist vereinfacht dargestellt, um die Beteiligung der implementierten Klassen zu veranschaulichen. Die Steuerung, die durch das Framework im Hintergrund erfolgt, wurde aus Übersichtlichkeitsgründen nicht dargestellt. Analog dazu laufen auch die Anwendungsfälle „Änderung der Benutzer-Einstellungen“ und „Löschen eines Benutzers/ einer Benutzerin“ ab. Statt *NewUserController* wird dann entweder *EditUserController* oder *DeleteUserController* eingesetzt. Auch die JSP-Seite ist in diesen Fällen eine andere als die Abgebildete.

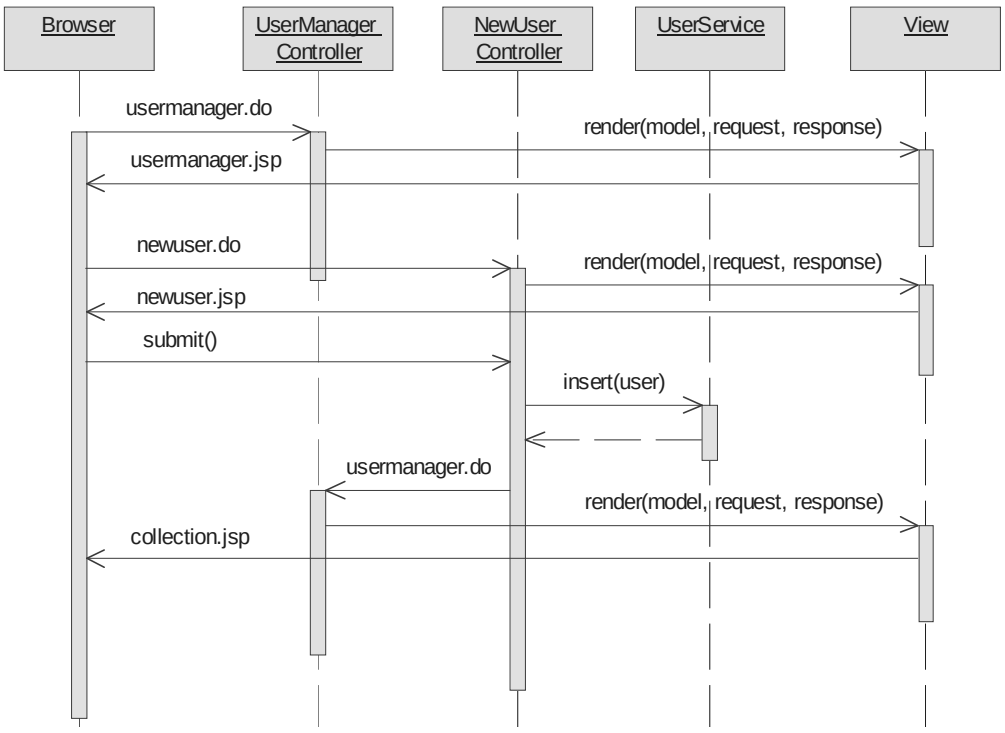

Abb. 10-35: Anwendungsfall: Anlegen eines euen Benutzer/ einer neue Benutzerin

Die Anfrage *“usermanager.do”* des Browsers gelangt zum *UserManagement*Controller, der das Model für die *usermanager.jsp* zur Verfügung

stellt, die im Browser angezeigt wird. Der/die AdministratorIn bekommt nun eine Übersicht über alle vorhandenen Benutzer-Accounts.

Er/sie wählt den Link „neuen Benutzer-Accounts anlegen“ aus, der in der Anfrage *“newuser.do”* resultiert. Diese Anfrage bearbeitet der *NewUserController*, der zuerst veranlasst, dass die *newuser.jsp*-Seite im Browser dargestellt wird. Der/die AdministratorIn wählt den/die BenutzerIn aus und stellt die Rechte und den Gültigkeitsbereich im Formular ein und schickt es ab. Der *NewUserController* reagiert auf das *submit()* in dem er das an das Formular gebundene *User*-Objekt an die *insert()*-Methode des *UserServices* übergibt, um es in die Datenbank zu speichern.

Anschließend wird die Kontrolle wieder an den *UserManagementController* zurückgegeben, der erneut alle bestehenden Benutzer-Accounts inklusive dem neu Angelegten anzeigt. Der Anwendungsfall ist somit beendet.

10.4.2 Erfassung

Der/die ErfasserIn bekommt eine Übersicht über alle von ihm erfassten, noch nicht freigegebene Forschungsarbeiten zu sehen, nachdem er/sie sich am System angemeldet hat. Er/sie hat nun die Möglichkeit eine Forschungsarbeit anzulegen, zu bearbeiten oder zu löschen. Das Anlegen und Bearbeiten wird von einem Controller übernommen, da das Formular das gleiche ist. Zum Löschen ist ein eigener Controller notwendig.

Es wird eine Forschungsarbeit an das Formular gebunden. Ist die CommandBean noch leer, dann ist der Controller im „Anlegen“ –Modus, sind hingegen schon Attribute mit Werten belegt, werden diese in den Formularfeldern angezeigt. Diese Funktionalität muss nicht implementiert werden, denn sie wird vom Framework geliefert.

Abb. 10-36 zeigt den Ablauf des Anwendungsfalls „Forschungsarbeiten anlegen“ in einem Sequenz-Diagramm, das jedoch nur die systemspezifischen Klassen enthält und auf die Darstellung frameworkinterner Klassen verzichtet:

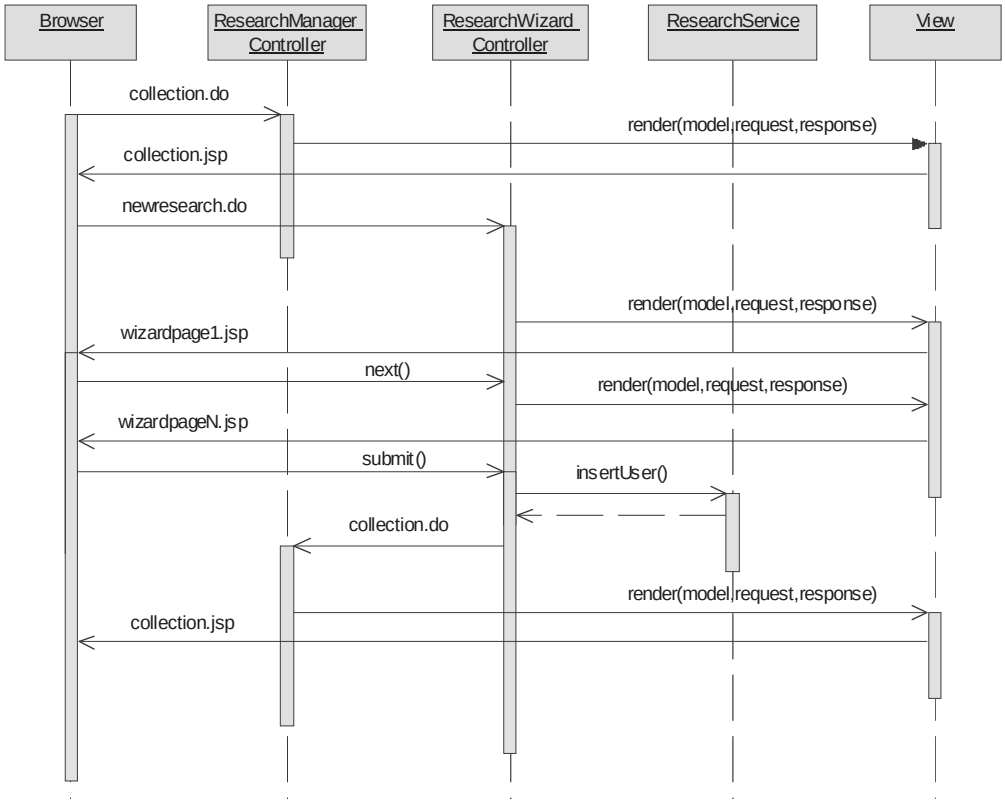

Abb. 10-36: Ablauf des Anwendungsfalls: Forschungsarbeiten erfassen

Wie Abb. 10-36 zeigt, beginnt der Anwendungsfall mit einer Anfrage *„collection.do“* des Browsers, die von dem *ResearchManagerController* verarbeitet wird. Von diesem wird die Seite *„collection.jsp“* an den Browser zurückgegeben. Durch die Anfrage *„newresearch.do“* gelangt die Steuerung des Requests an den *ResearchWizardController*, der dem Browser der Reihe nach die Wizard-Seiten mit den dazugehörigen Daten liefert. Sind alle Seiten angezeigt und ist das Formular abgeschickt worden (*submit*), wird die *insert*-Methode des *ResearchServices* aufgerufen um die neu angelegte Forschungsarbeit in die Datenbank zu speichern. Danach wird die Kontrolle wieder an den *ResearchManagerController* übergeben, der dann die aktualisierte Übersicht an den Browser liefert.

Da die Menge der Daten, die eingegeben werden muss, sich nicht auf einer Seite darstellen lässt, ist ein Wizard-Formular erforderlich. Das an dieses Formular gebundene Forschungsarbeit wird mit jeder Wizard-Seite zu einem Teil befüllt. Hat der/die ErfasserIn die letzte Seite erreicht, sollten alle erforderlichen Attribute der Forschungsarbeit gesetzt worden sein. Der *ResearchWizardController* erweitert die Framework-Klasse *AbstractWizardFormController*. So ist es nicht notwendig, den Kontrollfluss von einer Wizard-Seite zur nächsten zu steuern, denn das macht die Oberklasse. Dazu werden dem Controller alle Seiten des Wizards der Reihe nach in

einem Array angegeben. Ist beim Submit ein Formularfeld *_targetX* auf *true* gesetzt, wird im Browser die Seite angezeigt, die X als Indexwert des Seiten-Arrays hat. Da für diesen Prototyp Grafiken als „Weiter“ -Schaltflächen verwendet wurden, wird mit Hilfe von JavaScript-Funktionen ein Formularfeld mit dem entsprechenden *_targetX* Namen gesetzt und die Seite abgeschickt. Abb. 10-37 zeigt diese JavaScript-Funktionen:

```
/*set formfield to targetX true*/
goToPage = function (targetPage) {
   setSubmitFieldProperty("_target"+targetPage, "true");
  document.mainform.submit();
}
/*set form field value*/
setSubmitFieldProperty = function (name, value) {
   var formField = document.forms.mainform.submitField
  formField.name = name;
  formField.value = value;
}
```

Abb. 10-37: JavaScript Funktionen zum Setzen des Target-Parameters

Ansonsten bietet der *AbstractWizardFormController* die gleichen Annehmlichkeiten wie der *SimpleFormController*. Die *referenceData()*-Methode wird automatisch aufgerufen, bevor die Seite angezeigt wird. So können die Daten, die dargestellt werden müssen der Seite zur Verfügung gestellt werden. Abb. 10-38 zeigt einen Ausschnitt dieser Methode:

```
protected Map referenceData(HttpServletRequest request,
             Object command, Errors errors, int page) {
   […]
   List studies = studyService.getAll();
   model.put("studies", studies);
   List departments = departmentService.getAll();
   model.put("departments", departments);
   […]
   return model;
}
```

Abb. 10-38: referenceData()-Methode des ResearchWizardControllers

Die *referenceData()*-Methode aus Abb. 10-38 holt sich mit Hilfe der entsprechenden Services die Daten aus der Datenbank, die zur Anzeige der Wizard-Seiten notwendig sind. Das Service *studyService* liefert eine Liste aller Studienrichtungen, die auf der Kunstuniversität Linz möglich sind. Diese Liste wird dann dem Model, realisiert durch eine HashMap, unter dem Schlüssel *“studies”* übergeben. Die JSP-Seite kann dann

über den gleichen Schlüssel dann auf die Liste zugreifen. Analoges gibt für die Organisationseinheiten der Kunstuniversität.

Statt der *onSubmit()*-Methode werden die beiden Methoden *processCancel()* und *processFinish()* implementiert. Wie die Namen schon vermuten lassen, soll die erste der beiden Methoden aufgerufen werden, wenn der/die ErfasserIn die „Abbrechen“ - Schaltfläche benutzt, die zweite wenn „Speichern“ gedrückt wird. Auch diese Steuerung übernimmt das Framework, indem die Auswahl der Methoden davon abhängig gemacht wird, ob ein Formularfeld mit dem Namen *_cancel* oder mit *_finish* beim Submit auf *true* gesetzt ist. Abb. 10-39 zeigt einen Code-Auszug der beiden Methoden:

```
protected ModelAndView processFinish(
HttpServletRequest request, HttpServletResponse response,
 Object command, BindException errors) throws Exception {
   Exhibition ex = (Exhibition) command;
   […]
   if (ex.getId() == 0) {
      exService.insert(ex);
   } else {
      exService.update(ex);
   }
   response.sendRedirect("collection.do");
   return null;
}

protected ModelAndView processCancel(
 HttpServletRequest request, HttpServletResponse response,
 Object command, BindException errors) throws Exception {
   response.sendRedirect("collection.do");
   return null;
}
```

Abb. 10-39: Speichern und Abbrechen Methoden des Wizards

Wie in Abb. 10-39 ersichtlich ist, ist eine Forschungsarbeit an das Eingabe-Formular gebunden. Dieses wird durch den Parameter *command* den Methoden übergeben. Diese CommandBean wird in der *processFinish(...)*-Methode einem *Exhibition*-Objekt zugewiesen und die Id des Objekts wird überprüft. Ist diese ungleich Null existiert die Ausstellung schon und es wird ein Update ausgeführt. Im anderen Fall wird die *insert()*-Methode des *ExhibitionServices* aufgerufen. Wurde die Ausstellung gespeichert, wird dem/der ErfasserIn wieder die Übersicht angezeigt, in der nun die „neue“ Ausstellung ebenfalls aufscheint. Die *processCancel(....)*-Methode leitet sofort an die Übersicht weiter, da das Objekt nicht gespeichert werden soll, wenn „Abbrechen“ gewählt wurde.

10.4.3 Kontrolle

Da das praktische Projekt dieser Diplomarbeit nur ein Prototyp ist, sind die Controller für die Kontrolle und Freigabe von Forschungsarbeiten noch nicht implementiert. Es wird somit im Folgenden die geplante Vorgehensweise bei der Realisierung dieser Controller erläutert.

Nach der Anmeldung am System bekommt der/die KontrolleurIn eine Übersicht über alle Forschungsarbeiten, die noch nicht freigegeben worden sind und die einer der Organisationseinheiten zugeordnet sind, für die er/sie berechtigt ist. Ausgehend von der Übersicht kann der/die KontrolleurIn eine Forschungsarbeit bzw. ein Forschungsobjekt auswählen und bekommt dieses angezeigt. Empfindet der/die KontrolleurIn die eingegebenen Daten als korrekt, gibt er/sie diese mittels geeigneter Schaltfläche frei und gelangt wieder zur Übersicht, in der die freigegebene Forschungsarbeit nicht mehr aufscheint.

In Abb. 10-40 wird der Anwendungsfall „Forschungsarbeiten kontrollieren“ noch einmal in einem Sequenz-Diagramm dargestellt. Dieser Ablauf ist vereinfacht dargestellt, da auf die Darstellung der Framework-Steuerung verzichtet wurde:

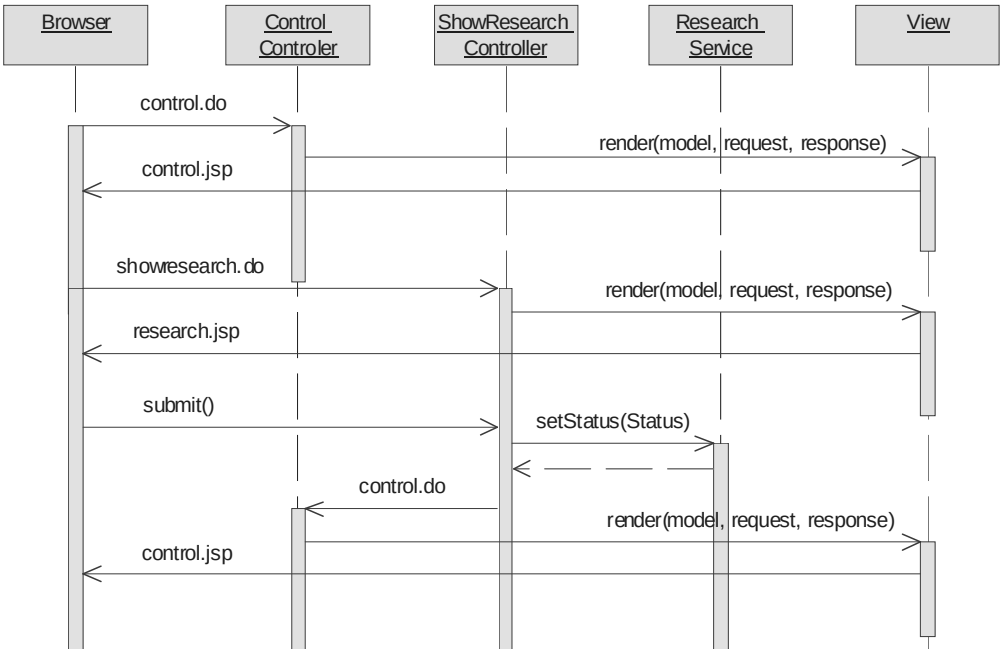

Abb. 10-40: Ablauf des Anwendungsfalls: Forschungsobjekt kontrollieren

Wie in Abb. 10-40 zu sehen ist, wird die Anfrage “*control.do*” des Browsers von der Klasse *ControlController* verarbeitet. Dieser Controller ist für die Darstellung der Übersicht über die freizugebenden Inhalte zuständig. Er stellt die Daten zur

Verfügung, die die JSP-Seite *controlmanager.jsp* zur Anzeige der Forschungsarbeiten benötigt. Durch die Auswahl eines Inhaltes veranlasst der/die KontrolleurIn eine Anfrage *"showresearch.do"*, die an den *ShowResearchController* gerichtet ist. Dieser ist ein Formular-Controller, da der/die KontrolleurIn die Forschungsarbeit durch eine geeignete Schaltfläche freigeben kann. Durch das *submit()* der *research.jsp*-Seite ruft der *ShowResearchController* die Methode *setStatus(…)* der Klasse *ResearchService* auf, damit die *status*-Variable der Forschungsarbeit in der Datenbank aktualisiert wird und der Inhalt öffentlich verfügbar wird.

11. Kapitel

Ausblick

In diesem Kapitel werden zum Abschluss die Entwurfsentscheidungen dieser Diplomarbeit reflektiert. Anschließend werden offene Punkte aufgezeigt und die geplante weitere Vorgangsweise im mm:widok Projekt geschildert.

11.1 Kritische Würdigung

Durch die Verwendung des Spring Web-Frameworks wurde die Implementierung der Präsentationslogik erheblich erleichtert, denn im Zuge der Wizard-Formular-Unterstützung wird automatisch an die nächste Wizard-Seite weitergeleitet und trotzdem ist die Information, welche Seite gerade aktiv ist bzw. von welcher Seite die Benutzereingaben stammen, verfügbar.

Trotz des hohen Einarbeitungsaufwands, um die Konfigurationsmöglichkeiten des Frameworks und somit auch der Applikation zu verstehen und anwenden zu können, ist der Einsatz des „Inversion of Control"-Mechanismus durchaus positiv zu bewerten, da die gesamte Anwendung durch die somit realisierte lose Kopplung der Komponenten konfigurierbar und sehr flexibel geworden ist.

Die deklarative Transaktionsunterstützung gestaltete den sonst – falls die Architektur sich nicht auf EJBs stützt – aufwendigen Umgang mit Transaktionen einfach.

Die Verwendung des JDBC-Abstraction-Frameworks hat den Datenbankzugriffscode um vieles weniger fehleranfällig gemacht und, obwohl einige Code-Zeilen zusätzlich notwendig waren, wurde dadurch Entwicklungszeit gespart.

Da der Großteil der Arbeit in der Speicherung und dem Auslesen komplexer Objekte bestand, wäre es im Nachhinein betrachtet besser gewesen, ein „Object-Relational-Mapping-Tool“ [Barr04], [John04], [Tura03] zu verwenden. Dazu wäre jedoch eine erneute Evaluierung und Einarbeitung notwendig gewesen.

11.2 Offene Punkte

Um aus dem im Rahmen dieser Diplomarbeit erstellten Prototyps ein vollständig einsetzbares System zu entwickeln wären noch folgende Tätigkeiten notwendig:

Bei der Vorführung des Prototyps, wurde von Seiten der Kunstuniversität erkannt, dass die Anforderung an die Zuordnung der Benutzerrechte, so wie sie am Projektbeginn erarbeitet wurde, nicht ausreichend ist. Es sei besser jeder Rolle eine oder mehrere Organisationseinheiten zuzuordnen, anstatt jedem/ jeder BenutzerIn für eine Organisationseinheit mit samt ihrer hierarchischen Untereinheiten zu autorisieren. Um diese erneute Anforderung zu verwirklichen wären Änderungen in allen Schichten der Applikation notwendig.

Die Funktionalität der Erfassung muss auf die anderen Arten von Forschungsarbeiten und Forschungsobjekten erweitert werden, da es derzeit nur möglich ist, Ausstellungen und Bilder repräsentativ für Forschungsarbeiten bzw. -objekte zu erfassen. Außerdem muss das Design und Layout dieser Seiten, wie auch der Administrationsseiten überarbeitet werden.

Der zu Beginn des Projekts erarbeitete, im Prototyp umgesetzte Wizard-in-Wizard Ansatz – mit dem Forschungsobjekte innerhalb von Forschungsarbeiten erfassen werden – sollte noch einmal überarbeitet werden. Die Forschungsobjekte könnten getrennt erfasst und erst im Zuge der Erfassung der Forschungsarbeiten referenziert werden. Zur Realisierung dieser Möglichkeit wären keine Änderungen in der Geschäftslogik notwendig. Es würde jedoch Änderungsaufwand in der Präsentationslogik sowie im Design der Seiten erforderlich sein.

Die vorgesehene Kontrollfunktion muss noch auf Präsentationsebene implementiert werden, das bedeutet die Implementierung eines Controllers und die Gestaltung der dazu gehörenden Webseite.

Die Suche nach Forschungsarbeiten und -objekten auf Basis verschiedener Metadaten muss implementiert werden und es muss eine entsprechende Benutzerschnittstelle dafür zur Verfügung gestellt werden. Laut Anforderungsbeschreibung soll diese in Flash realisiert werden. Die Java-Objekte der Suchergebnisse müssen in XML-Dokumente transformiert werden, einerseits damit Flashmethoden darauf zugreifen können und andererseits um damit die Integration mit anderen (inter)nationalen Forschungs- und Wissensdokumentationen zu ermöglichen. Dazu könnte die XML-Erweiterung von Oracle verwendet werden, die auch für die Realisierung der FoDok der JKU herangezogen wurde (siehe [Wies04]).

Das gesamte System muss in die Applikations- und Serverlandschaft der Kunstuniversität Linz integriert werden. Dort muss die Interaktion sowohl mit dem LDAP- als auch mit dem Real Media-Server getestet werden. Außerdem müssen entsprechende Views auf die bereits vorhandenen und von der mm:widok mitbenutzten Datenbank-Tabellen erstellt, integriert und getestet werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Anas01] Michalis Anastopoulos,Tim Romberg, Referenzarchitekturen für Web-Applikationen, Dezember 2001
http://www.fzi.de/KCMS/kcms_file.php?action=link&id=141

[Apac04] Apache Software Foundation, Dezember 2004
http://www.apache.org/

[Barr04] Object-Relational Mapping Article and Products,
Barry Assosiates, Burnsville, USA, Dezember 2004
http://www.service-architecture.com/object-relational-mapping/

[DCMI03] Dublin Core Metadaten Initiative, Dublin Core Metadaten Set, Version 1.1, Juni 2003
http://dublincore.org/,

[Eagl04] Mark Eagle, Wiring Your Web Application with Open Source Java, Juli 2004
http://www.onjava.com/lpt/a/4744,

[Ford04] Neal Ford, Art of Java Web Development, Manning Publications, Greenwitch, 2004

[Free04] FreeMaker Java Template Engine, Version 2.3, Juni 2004
http://freemarker.sourceforge.net/

[Howa04] Howard M. L. Ship, Tapestry in Action, Manning Publications, Greenwitch, 2004

[Hust02] Ted Husted, Cedric Dumoulin, George Franciscus, David Winterfeldt, *Struts in Action*, Manning Publications, Greenwitch, 2002

[John03] Rod Johnson, expert-one-on-one J2EE design and development, wrox, Wiley Publishing, Indianapolis, 2003

[Macr04] Macromedia Flash MX, Dezember 2004
www.macromedia.com

[Mave04] Maverick, Version 2.2 , Juni 2004
http://mav.sourceforge.net/

[McLa00] Brett McLaughlin, Web Publishing Frameworks, Juni 2000
http://www.oreilly.com/catalog/javaxml/chapter/ch09.html

[Nove04] Novell LDAP Server, eDirectory 8.7.3, Dezember 2004
http://www.novell.com/

[Ognl04] Object-Graph-Navigation-Languge OGNL, Version 2.6.5, April 2004
http://www.ognl.org/

[Orac03] Oracle Database 9i, September 2003
http://www.oracle.com/index.html

[Real04] Real, Helix Server, Dezember 2004
http://www.realnetworks.com/products/media_delivery.html

[Spri04] Springframework, SourceForge Project, Version 1.1, September 2004
http://www.springframework.org/

[Stru04] Apache Jarkata Project, Struts, Version 1.2, August 2004
http://struts.apache.org/index.html

[Sun00] Sun, Java Transaction API, Juni 2000
http://java.sun.com/products/jta/

[Sun03a] Sun, Java Server Pages, Version 2.0, April 2003
http://java.sun.com/products/jsp/

[Sun03b] Sun, Java Servlets, Version 2.4, November 2003
http://java.sun.com/products/servlets/

[Sun04a] Sun, Java Server Pages Standard Tag Library, Version 1.2, Juni 2004
http://java.sun.com/products/jsp/jstl/

[Sun04b] Sun, Core J2EE Pattern, Juli 2004
http://java.sun.com/blueprints/corej2eepatterns/Patterns/

[Tape04] Apache Jarcata Project, Tapestry Framework, Version 3.0, April 2004
http://jakarta.apache.org/tapestry/index.html

[Tape04w] Apache Jarcata Project, Wiki Seite des Tapestry Frameworks
http://wiki.apache.org/jakarta-tapestry

[Tho03a] Kris Thompson, Introduction to Maverick, August 2003
http://www.theserverside.com/articles/article.tss?l=Maverick

[Tho03b] Kris Thompson, Building with WebWork2, November 2003
http://www.theserverside.com/articles/article.tss?l=WebWork2

[Thom04] Kris Thompson, Web-Framework User Group, Dezember 2004
http://www.frameworks-boulder.org/Application_Frameworks.html

[Tura03] Volker Turau, Krister Saleck, Christopher Lenz, Web-basierte Anwendungen entwickeln mit JSP2, dpunkt.verlag, Heidelberg, 2003

[Velo04] Apache Jarkarta Project, Velocity Template Engine, Version 1.4, April 2004
http://jakarta.apache.org/velocity/

[Wafe04] Open Source Web-Framework Comparison Project, 2003
http://waferproject.org/index.html;

[Webw04] OpenSymphony Group, Open Source Web-Framework, WebWork2 Version 2.1, 2004
http://www.opensymphony.com/webwork/wikidocs/ Documentation.html

[Webw04w] OpenSymphony Group, Wiki-Seite des Open Source Web-Frameworks WebWork2, Version 2.1, 2004
http://wiki.opensymphony.com/display/WW/WebWork

[Whis04] Overview and Comparison of Java-Web-Frameworks, Juni 2004
http://213.190.42.244/whisper/space/Web+Frameworks

[Wies03] Herwig Wiesinger, Reengineering der Forschungsdokumentation der JKU Linz, November 2003, Diplomarbeit an der Johannes Kepler Universität Linz

[W3C99] XSL Transformation Version 1.0, W3C Recommendation, November 1999
http://www.w3.org/TR/xslt

[W3C00] Document Object Model Level 3, W2C Recommendation, April 2004
http://www.w3.org/DOM/