M A G I S T E R A R B E I T

zur Erlangung es akademischen Grades

Diplom Ingenieur

der Studienrichtung

Medizinische Informatik

EINSATZ DES MODEL-VIEW-CONTROLLER-KONZEPTS BEI JAVA-BASIERTEN WWW-ANWENDUNGEN

am Beispiel von Java Server Pages, Struts und Java Server Faces

ausgeführt am

Institut für Med. Bildverarbeitung und Mustererkennung

der

Medizinischen Universität Wien

Unter der Anleitung von

Univ.-Prof. DI Dr. Ernst Schuster

mitbetreuender Assistent

Dipl.-Ing. Georg Fischer

eingereicht von

Harald Kristoffer Kornfeil
Matrikelnummer 9125885

…… ……

(Ort, Datum) (Harald Kornfeil)

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

…… ……

(Ort, Datum) (Harald Kornfeil)

Diese Magisterarbeit ist in großer Dankbarkeit meiner Frau

Silke Martini

gewidmet.

An dieser Stelle sei auch all meinen Freunden, meinen Kollegen und den Professoren gedankt, die mich während meines Studiums begleitet und mich auf verschiedenste Art und Weise unterstützt haben.

Kurzfassung

In frühen interaktiven WWW-Anwendungen waren Programmcode und HTML-Anweisungen in den einzelnen Programmdateien untrennbar miteinander vermischt. Die Tatsache, dass heute vermehrt Wert auf anspruchsvolles WWW-Design gelegt wird sowie die zunehmenden Komplexität moderner WWW-Anwendungen mit immer mehr Beteiligten lässt einen solchen Lösungsansatz nicht mehr zu. Die Programmierung gemäß dem Model-View-Controller Konzept (MVC) basiert auf der Trennung von Datenverarbeitung, Darstellung und Anwendungslogik. Dadurch wird es möglich, dass Programmierer und Webdesigner gemeinsam an einem Projekt arbeiten können, Logik und Darstellung aber voneinander getrennt bleiben. Kleine, auf dem MVC-Konzept basierende WWW-Projekte lassen sich in Java mit JSPs, Servlets und mit Hilfe der Standardbibliotheken erstellen. Für die Realisierung größere Projekte werden moderne Java-Frameworks wie Struts und JavaServer Faces benötigt. Sie basieren vom grundlegenden Design her schon auf dem MVC-Konzept aber verwirklichen es auf unterschiedliche Weise.

Abstract

In early WWW applications program-code and HTML instructions were mixed inseparable in single program files. The demand of appealing designed WWW pages and the increasing complexity of modern WWW applications with more and more people involved do not allow such an approach any more. The Model-View-Controller Concept (MVC) is based on the separation of data processing, presentation and application logic. This makes it possible that programmers and web designers can jointly work on the same project without interfering with each other - leaving logic and presentation separated. Small MVC based WWW projects can be implemented in Java with JSPs, Servlets and the standard libraries. For bigger projects modern Java frameworks like Struts and JavaServer Faces are needed. They are MVC based by design but their approach is different.

Inhalt

1. EINLEITUNG – FRAGESTELLUNG 1

Probleme bei der Realisierung von web-basierten Anwendungen 1

2. GRUNDLAGEN 5

2.1. Java-Server 6

2.2. Web Application 7

2.3. Servlets 9
Allgemeines 9
RequestDispatcher 14
Forward 14
Include 15
Sitzungsmanagement 20

2.4. JavaServer Pages – JSP 24
Allgemeines 24
Skripte 27
Anweisungen – Scriptlets 27
Ausdrücke – Expressions 29
Vereinbarungen – declarations 29
Kommentare – comments 31
Direktiven - directives 31
page-Direktive 31
include-Direktive 34
taglib-Direktive 36
Aktionen - actions 47
Forward-Aktion 47
Include-Aktion 49
JavaBeans-Aktionen 52
MVC mit JSP 59

2.5. Java Standard Tag Library - JSTL 61
Expression Language – EL 62
Function Tag Library 64
Die Core-Tag-Library 64
Deklaration 65
Variablen 65
Schleifen 65
Bedingungsabhängige Programmflusssteuerung 66
Einfügen von Text 67
Request-Umleitung 67
MVC mit JSP und JSTL 68

3. METHODIK 70

3.1. Aufgabenstellung: 70

3.2. Entwurf 71
Model 71
Controller 72

4. ERGEBNISSE 75

4.1. Implementation gemeinsamer Komponenten 75
Model 75
Web Application Deployment Discriptor `web.xml` 77

4.2. Implementation der drei MVC-Beispiele 78
Controller und Views in der JSP/JSTL-Version 78
Controller und Views in der Struts-Version 86
Controller und Views in der JSF-Version 99

5. DISKUSSION 115

5.1. Allgemeines 115
Performance 115
Features 116

5.2. Model 116

5.3. Controller 118

5.4. View 119
Tags und EL 119
Darstellung 119
Barierrefreiheit 120
Internationalisierung (i18n) 120

6. ZUSAMMENFASSUNG – SCHLUSSFOLGERUNGEN – AUSBLICK 120

7. VERZEICHNISSE 122
Abbildungsverzeichnis 122
Tabellenverzeichnis 122
Beispiele 123

8. ANHANG 129

Anhang 1 Methodenübersicht der Model-Beans 130
Aufgabe 130
Aufgabenkatalog 131

Testverlauf 131

Anhang 2 Vollständige Programmcodes der Beispiel-Web-Anwendungen 133
Session-Beispiel 133
Model-Beans inkl.ServletContext-Listener SCInit 135
JSTL-Version der Knoko-Lernprogramm-Webapplikation 140
Struts-Version der Knoko-Lernprogramm-Webapplikation 144
JSF-Version der Knoko-Lernprogramm-Webapplikation 149

Anhang 3 Dateiformat der Textdatei knokofragen.txt 153
Dateiformat 153
Beispiel 153

1. Einleitung – Fragestellung

Diese Arbeit untersucht wie moderne Java-basierte Frameworks das Model-View-Controller-Konzept (MVC) in die Praxis umsetzen. Worum es sich beim MVC-Konzept handelt, wieso es in den letzten Jahren bei der Realisierung von Web-Anwendungen vermehrt zur Anwendung gekommen ist und welche Probleme man bei der Entwicklung damit beseitigen will zeigt am besten die geschichtliche Evolution von interaktiven (Java-basierten) Web-Anwendungen.

Probleme bei der Realisierung von web-basierten Anwendungen

Um Internet-Benutzern über das World Wide Web neben statischem HTML-Seiten auch interaktive Datenabfragen zur Verfügung stellen zu können, wurden WWW-Anwendungen anfangs ausschließlich serverseitig mit Hilfe des Common Gateway Interface (CGI) oder durch Verwendung von Server-Side-Includes (SSI) der Web-Servers realisiert.
CGI-Programme können in nahezu jeder beliebigen Programmiersprache realisiert werden, da das Funktionsprizip sehr einfach ist. Bei jeder Anfrage wird ein CGI-Programm vom Web-Server gestartet und es erhält die an den Web-Server übermittelten Daten von diesem über stdin und schickt seine Ausgabe über stdout an den Webserver zurück, welcher die Ausgabe an den Browser zur Darstellung weiterleitet.
Bei SSIs werden Befehle an den Server in den HTML-Code eingebettet und in Dateien mit der Datei-Extension .SHTML abgespeichert. Diese Dateien werden vom Server ausgewertet, bevor er die Web-Seite an den Client liefert. Auf diese Weise können Variablen angezeigt, Dateien eingefügt oder externe Programme gestartet und deren Ausgabe eingefügt werden. Die Syntax der Kommandos ist sehr einfach. Sie sind in HTML-Kommentare eingebunden.
Solange die (CGI-)Programme mit einer Interpretersprache (z.B. Perl) geschrieben sind, sind die WWW-Anwendungen plattformunabhängig. Werden sie jedoch in einer Compilersprache (z.B. C) verfasst, muss der Quellcode bei einem Plattformwechsel neu übersetzt werden.
Da beim Aufruf von externen Programmen jedes Mal ein Programm neu gestartet werden muss, wurde nach Lösungen gesucht die den Scheduler des Betriebsystems, auf dem der Webserver läuft, nicht so stark belasten.
Eine dieser Lösungen ist FastCGI. FastCGI-Programme werden nach ihrer Nutzung nicht beendet und im Speicher behalten. Eine ähnliche Lösung für Perl-Programme ist mod_perl für den Apache Web-Server. Die Web-Server von Microsoft und Netscape führten eine weitere Möglichkeit ein um externe Programme einzubinden: Microsofts Internet Information Server die ISAPI und der Netscape Enterprise Server die NSAPI. Dabei handelte es sich um ein Application Programming Interface (API) welches zur Erstellung von dynamic Link Libraries (DLL) diente. Diese wurden bei einem Request dynamisch eingebunden und ausgeführt. Aufgrund der engen Integration in den Web-Server sind die Programme zwar schneller, damit aber plattformabhängig.

Die Realisierung einer interaktiven WWW-Anwendung über den Aufruf von externen Programmen und DLLs bringt einige Nachteile mit sich. Ein Problem ist, dass der statische und dynamische Teil einer WWW-Anwendung am Server in verschiedenen Verzeichnissen voneinander getrennt sind. Die CGI-Programme sind im ausführbaren CGI-BIN-Verzeichnis gespeichert, die statischen Seiten zusammen mit anderen HTML-Seiten im nicht ausführbaren HTML-Verzeichnis. Ein viel schwerwiegenderer Nachteil ist, dass die Ausgabe der dynamischen Seiten über eine Vielzahl von print-Anweisungen realisiert werden muss, bzw. dass der auszugebende Text und die zugehörigen Tags im Programmcode verteilt sind.
Das Schreiben eines CGI-Programms ist vom Prinzip her zwar einfach, das Schreiben eines *sicheren* CGI-Programmes ist jedoch nicht trivial. Vom Sicherheitsstandpunkt ist es schwierig einen Web-Server so abzusichern, dass die externen Programme daran gehindert werden auf

Dateien und Ressourcen des Web-Servers zuzugreifen. Die Programme könnten darüber hinaus sogar Exploits enthalten die es ermöglichen könnten auf dem Web-Server Hintertüren und Rootkits zu installieren. Der Administrator eines Webservers hat daher nur die Wahl einem Programm blind zu vertrauen oder den Programmcode zu überprüfen. Beide Optionen sind nicht zufrieden stellend.

Die ersten beiden Punkte können durch einen Ansatz ähnlich dem Prinzip der SSI realisiert werden. Anstatt die HTML-Ausgabe in den Programmcode zu integrieren, wird in die HTML-Dateien Script-Code eingefügt. Bei einem Request wird die Datei vom Server analysiert und der in den HTML-Code eingebettete Script-Code ausgeführt. Der dynamisch generierte Inhalt erscheint sodann anstatt des Script-Codes an der entsprechenden Stelle im statischen Text und wird an den Client geliefert.
Technologien die auf diesem Prinzip beruhen sind PHP (PHP Hypertext Processor), ASP (Advanced Server Pages), ePerl und andere.

Im Jahr 1997 wurde von Sun Microsystems der „Java-Server“ (Codename Jeeves) veröffentlicht[1]. Hierbei handelte es sich um einen vollständig in Java programmierter Web-Server. Java wurde von Sun Anfang der neunziger Jahre im Rahmen des Projekts „Green“ ursprünglich zur Programmierung von Set-Top-Boxen, Videorecordern, Telefonen und anderer elektronischer Geräte entwickelt. Die Java-Programme sollten in einer auf diesen Geräten installierten, vom Prozessor unabhängigen, virtuellen Maschine ablaufen. Die Programme laufen dabei in einer „Sandbox“ aus der sie im Normalfall nicht ausbrechen können. Mit der Integration der Java virtual Machine (JVM) in den Navigator 2.0 der Firma Netscape wurden im Jahr 1996 die so genannten *Applets* eingeführt: kleine in Web-Seiten eingebettete Programme, die innerhalb des Internet-Browsers auf dem Client, vom Betriebssystem unabhängig, ausgeführt wurden. Durch den Sandbox-Mechanismus wird ein Übergriff eines Programms mit schädlichem oder fehlerhaftem Code auf das Betriebssystem unterbunden.
Ziel des Java-Servers war es eine plattformunabhängige Programmierung von WWW-Anwendungen zu ermöglichen. Zusammen mit dem Java-Server wurde auch das Java Server Toolkit veröffentlicht. Dieses stellt im Grunde ein API zur Programmierung interaktiver WWW-Applikationen dar. Die damit erstellten Servlets („Server-side-Applets“) werden, ähnlich der DLLs bei ISAPI/NSAPI, bei Aufruf dynamisch in dem Web-Server geladen.

Da die Servlets von der Technologie her dem DLL-Ansatz ähneln, teilen sie sich auch dieselben Probleme bezüglich der Ausgabe. Das Sicherheitsproblem, dass externe Programme Zugriff auf das Betriebssystem haben, wurde durch den Java-Server teilweise gelöst, da der Zugriff auf Ressourcen des Systems über die JVM eingeschränkt werden können.
Mit dem JavaServer 1.1 führte SUN die Servlet Beans und JHTML ein[2]. JHTML kann als eine Art Vorgänger der Java Server Pages (JSP) gesehen werden, unterscheidet sich jedoch grundsätzlich in der Syntax von JSP. JHTML-Dateien konnten in `<JAVA>`-Tags eingebetteten Java-Programmcode und Aufrufe von Servlet-Beans enthalten und wurden vom Java-Server beim ersten Aufruf übersetzt. Auch eine SSI-Funktionalität war vorhanden. In Dateien mit der Endung .SHTML wurden an den entsprechenden Stellen die `<SERVLET>`-Tags durch die Ausgabe des im Tag angegebenen Servlets ersetzt. ServletBeans, JHTML und das `<SERVLET>`-Tag werden heute kaum noch von Java-Web-Servern unterstützt.

Ende 1998 veröffentlichte Sun die Servlet API Specification 2.1 zuammen mit der JavaServer Pages Specification 0.92. Bei ersterer handelte es sich im Wesentlichen nur eine Beschreibung der API des Java Server Toolkits 2.0 und beinhaltete nur geringfügige Änderungen. Die Spezifikation konzentrierte sich darauf vorzuschreiben wie das Servlet API

zu implementieren sei. Zweitere war die erste freigegebene Spezifikation für JSP in der Art wie es sie heute noch gibt [3]. Im Herbst 1999 folgte die Java Server Pages Specification 1.0 [4]. Sie beschreibt die Syntax der in Java Server Pages (JSPs) verwendeten spezielle <% %>-Tags, welche Schlüsselwörter, Deklarationen oder Programmcode enthalten können und wie Java-Bean-Komponenten über Standard-Aktionen verwendet werden können.
Diese Spezifikationen schrieben jedoch nicht vor wie verschiedene Web-Anwendungen auf einem Server sauber voneinander getrennt werden sollten, bzw. wie eine Web-Anwendung (HTML-Seiten, Bilder, JSPs und Servlets) installiert und deinstalliert werden sollten. Dadurch lösten verschiedene Hersteller dies auf verschiedene Arten. Auch der in die JSPs integrierte Java-Code machte den HTML-Code sehr schnell unübersichtlich.

Die Servlet API Specification 2.2 führte den Begriff Web Application ein: A web application is a collection of servlets, html pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors[5]. Mit der JSP-Spezifikation 1.1 wurden die Vorgaben für die Programmierung von eigenen Tag-Bibliotheken (Custom Tag Libraries) veröffentlicht. Durch die Kombination von JSP-Tag-Libraries und Java-Beans kann der in JSPs enthaltene Script-Code auf ein Minimum reduziert, die Wiederverwendung von Komponenten erhöht und die Struktur, Lesbarkeit und Wartbarkeit verbessert werden.
Die auf diesen Spezifikationen aufbauende Referenzimplementation ist der Java-Server Tomcat in der Version 3 [6], welcher der Apache Software Foundation von Sun gestiftet wurde. Bei Tomcat 3 konnte der Server-Administrator erstmals über den Security-Manager (Access Controller) eine Servlet Sandbox definieren in der Servlets agieren dürfen. Weiters wurde es möglich einzelnen Servlets gewisse Rechte zu geben bzw. zu nehmen.

Die folgende Spezifikationen (Servlet API 2.3/JSP 1.2) brachten auf Seiten der Servlets die Einführung von *Filtern* und die Ereignisverarbeitung mittels *Listenern* und bei den JSPs eine Vereinfachung zur Erstellung eigener Tags [7].
Im Rahmen des Java Community Process wurde von der Apache Software Foundation eine weitere Lücke der JSP-Spezifikationen geschlossen: basisierend auf den Vorgaben der Arbeitsgruppe JSR052 wurde eine Java Standard Tag Library (JSTL) entwickelt. Sie stellt oft benötigte Funktionalitäten (Programmablaufsteuerungsmechanismen, SQL-Datenbank-Zugriff, XML-Manipulationen, Internationalisierung) in Form von Standard-Tags zur Verfügung. Weiters wurde für die Tags der JSTL die „Expression Language“ (EL) entwickelt, die einen einfachen Zugriff auf die Eigenschaften von Java-Beans ermöglicht. Um diesen einfachen Zugriff auch außerhalb JSTL-Tags zu ermöglichen wurde die EL mit der JSP 2.0 Spezifikation (auf Basis der Servlet API 2.4) als offiziellen Bestandteil der JSPs eingeführt.
Parallel zu den Standardisierungsbestrebungen von Sun wurden vorrangig im Rahmen von Open-Source-Projekten der Apache Software Foundation viele weitere Projekte ins Leben gerufen, die Web-Programmieren und Web-Designern die Arbeit erleichtern sollten.
Das Hauptaugenmerk wurde dabei auf die Trennung von Programmlogik und Bildschirmdarstellung gelegt, da eines der Hauptprobleme bei der Erstellung einer Web-Anwendung die Vermischung der Programmlogik mit der Bildschirmdarstellung ist. Entweder der Text und die HTML-Tags sind im Programmcode verstreut oder der Programmcode ist in den HTML-Seiten enthalten. Da bei der Web-Seiten-Gestaltung heutzutage ein sehr anspruchsvolles Design gefordert wird, müssen Web-Anwendungen von Web-Designern und Programmierern im Team gestaltet und programmiert werden. Schon in der JSP Specification 0.92 [8] hat man sich bei Sun offenbar über Rollen der Mitwirkenden an einem webbasierten Projekts Gedanken gemacht („Typical User Roles“) und wie man die Anwendungsentwicklung am besten trennen könnte („JavaServer Pages Access Model“). Die Autoren unterscheiden nach Kenntnissen und Fähigkeiten der Beteiligten.

Rolle	Kenntnisse		
	HTML	Java / JavaBeans	JavaScript
Web-Designer	Ja	Nein	Nein
Komponenten Programmierer	Ja	Ja	Nein
Dynamic-HTML-Programmierer	Ja	Nein	Ja
Servlet Programmierer	Ja	Ja	Nein

Tabelle 1-1
Mitwirkende bei webbasierten Projekten

Um diese Trennung zu erreichen bzw. zu vereinfachen soll die Programmlogik so weit als möglich auf wieder verwendbare Komponenten ausgelagert werden. Weiters schlagen die Autoren zwei Access Models vor: Model 1 und Model 2.

Model 1: Ein Client-Request wird direkt an eine JSP gerichtet, welche über Beans auf eine Datenbasis zugreift und mit den erhaltenen Daten den Response erzeugt.

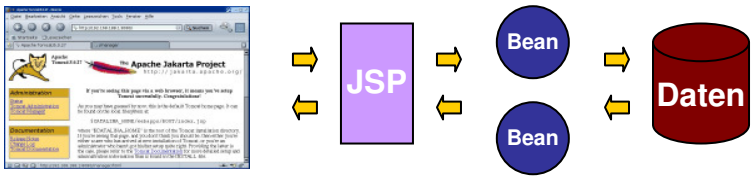

Abbildung 1-1
Model 1

Model 2: Ein Client-Request wird an ein Servlet gerichtet, welches selbst auf die Datenbasis zugreift und die erhaltenen Daten in einer JavaBean speichert. Die Response erzeugt eine vom Servlet aufgerufene JSP, welche die Daten durch die JavaBean erhält und damit die Response erzeugt.

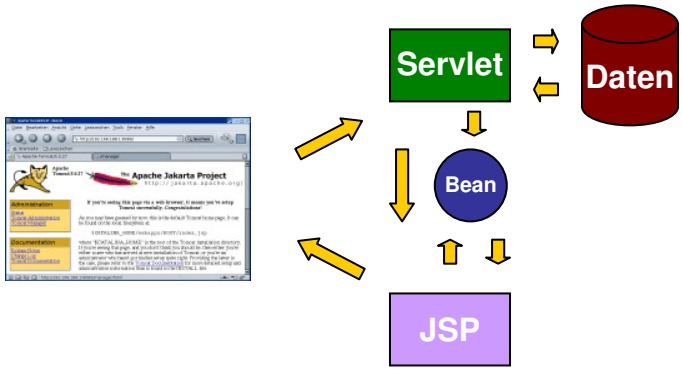

Abbildung 1-2
Model 2

Diese Vorschläge zur Architektur eine Servlet/JSP-basierten Web-Anwendung kommen in den weiteren veröffentlichten Servlet- und JSP-Spezifikationen nicht mehr vor, sind aber in

den J2EE Guidelines enthalten [9]. Das „Model 2“ wurde in einem Java-World-Artikel [10] aufgegriffen, wo der Autor anhand eines Beispiels zu zeigen versucht, dass das Model 2 die beste Möglichkeit ist Inhalt und Präsentation zu trennen.

Das Model 2 entspricht dem so genannten Model-View-Controller-Konzept (MVC). Es wird Trygve M. H. Reenskaug [11] zugeschrieben. Er beschäftigte sich Ende der 70er Jahre am Xerox Palo Alto Research Center (PARC) mit der Entwicklung fenster-basierter Benutzeroberflächen verteilter Systeme unter Smalltalk-80.
Das MVC-Konzept unterscheidet:

- Model - das „Datenmodell“ (in dem Informationen gespeichert sind) bzw. die so genannte „Geschäftslogik“.
- View - die visuelle Erscheinungsform des Programms nach außen („WWW-View“ für Browser oder „Fenster-View“ einer graphischen Benutzeroberfläche wie Java-Swing).
- Controller - die Logik, die entscheidet ob und wie auf bestimmte Ereignisse reagiert werden soll, greift lesend und schreibend auf das Model zu und entscheidet was in welchem View präsentiert wird.

Der Hauptvorteil dieser Form der Programmierung ist, dass sich das Aussehen und die Bedienung austauschen lassen, ohne das zugrunde liegende Datenmodell verändern zu müssen. Im Falle einer Web-Anwendung würde dies im Idealfall bedeuten, dass der Programmierer sich nicht darum kümmern muss wie die Web-Seiten aussehen werden und das Layout dem Webdesigner überlassen kann. Diese Ideen wurden im Mai 2000 aufgegriffen und das *Struts*-Projekt [12] [13] als Sub-Projekt des Jakarta Projekts der Apache Software Foundation ins Leben gerufen.

Doch auch die Entwicklung mit JSPs mit Tags und Beans sowohl mit als auch ohne Hilfe von WWW-Application-Frameworks setzt Grenzen bei der Wiederverwendung von (vor allem grafischen) Komponenten. Microsoft schuf im Rahmen der .net-Initiative eine komponenten-basierende Web-Entwicklung mit ASP.NET und Web-Forms. In Anlehnung an die GUI-Frameworks von Java-Applikationen (AWT, SWT, Swing) wurde im Frühjahr 2004 die JavaServer Faces Specification 1.0 (JSF) [14] veröffentlicht, die im Rahmen des Java Communitiy Process JSR 127 entwickelt wurde. JavaServer Faces [15] [16] basiert auf der Wiederverwendung von graphischen Komponenten und führt einen Steuerungsmechanismus über Ereignisbehandlung (Events) ein. Weiters wurde auch eine Referenz Implementation (RI) entwickelt auf deren Basis sich JSF-Anwendungen programmieren lassen. Von Open Source Communitiy wurde diese Technologie ebenfalls aufgegriffen und die Implementierung *MyFaces* entsprechend der JSF Spezifikationen als Projekt der Apache Software Foundation geschaffen [17] [18].

2. Grundlagen

Um besser verstehen zu können wie die Komponenten miteinander kommunizieren wird zunächst gezeigt wie die zugrunde liegenden Mechanismen eines JavaServers funktionieren.
Da die verschiedenen Programmiermodelle von JSTL, Struts und JavaServer Faces auf Servlets und JavaServer Pages aufbauen, werden anschließend ausführlich die Grundlagen der Servlet- und JSP-Programmierung erklärt.

2.1. Java-Server [19]

Der Begriff Java-Server bezeichnet nicht „serverseitiges Java“ im Allgemeinen. Serverseitiges Java kann auch mittels Enterprise-Java-Beans oder mittels Remote Method Invocation Mechanismen realisiert werden. Mit „JavaServer“ verbindet man vor allem serverseitige interaktive WWW-Anwendungen, die in Java geschrieben werden.
Der eigentliche JavaServer ist/war eigentlich der Java Web Server der bei Sun unter dem Codenamen „Jeeves“ als ein eigenes Produkt entwickelt wurde. Er unterstützt neben dem veralteten JHTML und Servlet-Beans nur Java Server Pages 1.0 und die Java Servlet-API 2.1. Die Weiterentwicklung dieses Produkts wurde eingestellt.
Sun Microsystems hat mit der Einführung des JavaServers und der Servlets jedoch nicht nur eine Möglichkeit geschaffen interaktive WWW-Anwendungen in Java zu schreiben. Das modulare Konzept des JavaServers in Verbindung mit Servlets kann im Prinzip für jede Art von Internet-Server-Anwendung verwendet werde. Als ein Beispiel sei der Java Apache Mail Enterprise Server (JAMES) der Apache Software Foundation genannt.

Oft werden für JavaServer auch die Bezeichnungen *Servlet-Engine* oder *Servlet-Container* verwendet.
Die Java Servlet Specification 2.1 [20] definiert eine *Servlet Engine* als „… a customized extension to a Web server for processing servlets, built in conformance with the Java Servlet API by the Web server vendor. The servlet engine provides network services, understands MIME-requests, and runs servlet containers. “. Sie hat die Aufgabe den *Servlet-Lifecycle* zu überwachen: Servlets zu laden, zu instanzieren und zu initialisieren. Wenn eine Client-Anfrage an ein Servlet bei der Servlet Engine eintrifft, übergibt sie dem Servlet ein ServletRequest-Objekt und ein ServletResponse-Objekt und lässt das Servlet ablaufen. Wird ein Servlet nicht mehr gebraucht, muss die Servlet Engine für eine geordnete Entfernung des Servlets sorgen.
Mit der Java Servlet Spezifikation 2.2 wird die Servlet API ein integraler Bestandteil der Java 2 Platform Enterprise Edition (J2EE). Da in der J2EE-Terminologie der Begriff Container schon etabliert ist, wird der Begriff Servlet Engine durch den Begriff *Servlet Container* ersetzt. Die Aufgaben eines Servlet Containers entsprechen weiterhin den oben angeführten Aufgaben[21].

Der im Rahmen dieser Arbeit vorgestellte Code wurde zur Gänze mit der Servlet Engine Tomcat 5 getestet. Tomcat ist eine Spende von Sun Microsystems and die Apache Software Foundation. Tomcat kann im Standalone-mode HTTP-Anfragen eines Clients verarbeiten und diese als HTTP-Antworten zurück schicken. Es ist aber auch möglich, dass Anfragen von einem vorgeschalteten Webserver an Tomcat weitergereicht werden, welcher sie verarbeitet und seine Antwort an den zwischengeschalteten Webserver schickt, welcher wiederum die HTTP-Antwort an den Client sendet. Letztere ist die von den Entwicklern empfohlene Methode, da die Servlet-Engine dann nicht so stark mit Anfragen an statische Inhalte belastet wird.

Die Servlet-Engine *JServ* des Apache-Java-Projects war nur zur Servlet API-Spezifikation 2.0 konform und wird nicht mehr weiterentwickelt. Die Apache-Entwickler empfehlen daher eine Migration auf Tomcat.

Weitere Servlet Engines[22] :

- Resin, Caucho Technologies, Open Source und komerziell.
- WebSphere-Application-Server, IBM
- JRun, Macromedia
- iPlanet WebServer, Sun/Netscape/AOL

- Oracle Application Server, Oracle
- ServletExec, NewAtlanta Communications

2.2. Web Application

Den Begriff „Web Application“ führte Sun mit der Java Servlet Specification 2.2 ein. Eine Web-Anwendung wird definiert als eine Ansammlung zusammengehörender Servlets, HTML-Seiten, Klassen und anderen Ressourcen welche in einem Paket gebündelt werden, und auf verschiedenen JavaServern unterschiedlicher Hersteller ablaufen können.
Da vor der Servlet Specification 2.2 nicht geregelt war wie mehrere auf einem Server liegende Web-Anwendungen von einander getrennt strukturiert gespeichert wurden, half man sich mit virtuellen Hosts. Die Entwickler der Apache Software Foundation unterschieden bei der Java-Server Toolkit 2.0 kompatiblen Servlet-Engine *Jserv* die Begriffe der *Servlet-zone* und *Servlet-Repository*.
Die JServ-Dokumentation[23] bietet dazu folgende Erklärung:

> Ein Servlet-Repository ist vergleichbar mit einem Verzeichnis - eine Servlet-zone vergleichbar mit einem virtuellen Host. Alle Servlet-Repositories, die in einer Servlet-zone enthalten sind, teilen einen gemeinsamen logischen Kontext.

Der „logische Kontext“ wurde schon bei der Servlet-API 1.0 definiert: als *ServletContext*. Mit der Definition der Web Application hat sich auch die Definition des ServletContexts geändert. In der Java Servlet Spezifikation 2.1 war der ServletContext noch „die Sicht des Servlets auf die Servlet Engine“. Seit Java Servlet Spezifikation 2..2 ist der ServletContext eingeschränkt als „die Sicht des Servlets auf die Web Application“. Ein ServletContext ist mit dem Verzeichnis der Web Application verwurzelt. Beispielsweise könnte sich ein ServletContext an der Adresse `http://servername/test` befinden. Jede Anfrage, deren Pfad mit `/test` (dem *Context-Pfad*) beginnt, wird von der Servlet Engine dem ServletContext „test“ zugeordnet, welcher die Web Application „Test“ repräsentiert.
Zwischen Web Application und ServletContext besteht eine 1-zu-1-Beziehung: es steht immer ein ServletContext für eine Web-Applikation zur Verfügung.
Eine Webapplikation kann auch als „distributable“ definiert werden, das heißt sie kann auf mehrere virtuelle Maschinen verteilt sein. Dann gibt es eine Instanz pro Web-Applikation pro Virtueller Maschine. Nicht nur der ServletContext ist auf eine Web Application begrenzt. Auch beim Zugriff auf Beans ist der Gültigkeitsbereich (*Scope*) auf den Kontext der jeweiligen Web Application beschränkt.

Web Applikationen werden bei Tomcat im Ordner *webapps* gespeichert. Aus der Sicht des Servers besteht eine Web Application aus der eigentlichen Anwendung (Dateien in einer Verzeichnishierarchie), einer Konfigurationsdatei und eventuell notwendigen zusätzlichen Bibliotheken. Die Wurzel dieser Verzeichnishierarchie ist zugleich auch die Wurzel für alle Dokumente (hierarchy-root = document-root = context-root). Nicht vollständige URLs („not fully qualified URLs“ = ohne Angabe von Protokoll, Servername usw.) z.B. in Hyperlinks von JSPs und HTML-Seiten beziehen sich auf den Ort an dem sie abgespeichert sind. Man nennt dies eine „relative“ Pfandangabe. Wird der Pfadangabe ein “/“ vorangestellt, dann handelt es sich um eine „context-relative“ Pfandangabe. Das heißt ein context-relativer Pfad bezieht sich immer auf die Wurzel der Web-Applikation (deren Basis im ServletContext-Objekt gespeichert ist) und nicht auf die Wurzel des Servers.
Innerhalb dieser Verzeichnishierarchie existiert ein spezielles Verzeichnis mit dem Namen „WEB-INF“. Da dieser Ordner per Definition nicht Teil des öffentlichen Dokumentenbaums ist, darf die Servlet-Engine keine unterhalb von WEB-INF gespeicherten Dateien direkt an einen Client liefern.

WEB-INF ist folgendermaßen strukturiert:

/WEB-INF/	hier befindet sich die Konfigurationsdatei der Web Applikation: der *Web Application Deployment Discriptor* web.xml. Weiters sind in diesem Verzeichnis auch TagLibrary-Deskriptoren (*.tld-Dateien) und andere Konfigurationsdateien zu finden.
/WEB-INF/classes	hier werden Servlet- und Hilfs-Klassen (Beans, Taglibrary-Klassen) abgelegt.
/WEB-INF/lib	hier werden für die Web Applikation notwendige JAR-Bibliotheken (Java-Archiv-Dateien, JDBC-Treiber, Taglibraries ...) gespeichert

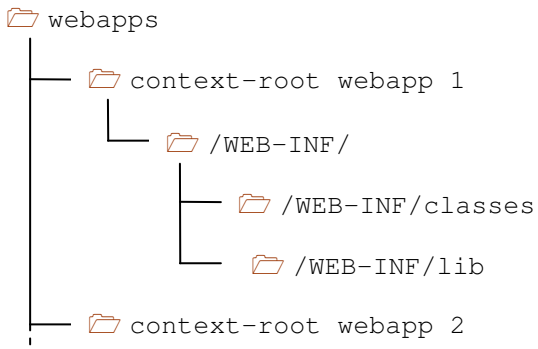

Abbildung 2-1
WEB-INF-Verzeichnisstruktur

Web Applikationen können in Web Application-Archive-Dateien (WAR) als Pakete zusammengefasst und signiert werden - analog zu Java-Archive-Dateien (JAR). Diese können dann ein /META-INF/ Verzeichnis enthalten, das - wie /WEB-INF/ - nach außen hin nicht sichtbar ist, und von Clients daher nicht abgefragt werden kann.
Diese Web Application Archives erlauben eine leichte Web-Applikations-Distribution, da die WAR-Dateien ganz einfach nur in das Verzeichnis webapps/ (siehe unten) kopiert und bei der Servlet-Engine angemeldet werden müssen. Letzteres geschieht entweder beim Start der Servlet-Engine oder über deren Administrations-Tools.

Die Datei web.xml, der Web Application Deployment Discriptor (auch *Web Application Configuration Discriptor*) enthält sowohl die Konfigurationsanweisungen der Web Application für die Servlet-Engine, als auch die Konigurationsanweisungen für die Web Applikation selbst. In einem XML-konformen Datenformat werden

- Init-Parameter für Context definiert
- Servlets registriert
- JSPs registriert
- Init-Parameter für Servlets und JSP definiert
- Servlet-Mappings definiert
- Tag-Libraries registriert
- Zugangsbeschränkungen definiert

Die Struktur von `web.xml` wurde in Servlet-Spezifikation 2.2 festgelegt und als Dokumententyp-Definition (DTD) veröffentlicht. In der Servlet Spezifikation 2.3 und 2.4 wurde sie erweitert. Die Version 2.4 des Deployment Descriptors existiert auch als XML-Schema. Im Rahmen dieser Arbeit werden nur auf die für die Programmbeispiele wichtigen Konfigurationsanweisungen beschrieben werden.

2.3. Servlets

Allgemeines [24]

Wie schon oben erwähnt können Servlets im Prinzip für verschiedenste Arten der serverseitigen Internet-Kommunikation verwendet werden. Da hier WWW-Anwendungen untersucht werden sollen, beschränken sich die Ausführungen im Folgenden auf die Aspekte der Kommunikation über das Hypertext Transfer Protokoll (HTTP). Dafür stellt die Servlet API eigene Klassen (javax.servlet.http) zur Verfügung, die von den generischen Servlet-Klassen abgeleitet sind (javax.servlet).

HTTP-Servlets sind plattformunabhängige Java Klassen die von einem Servlet Container eines Web-Servers dynamisch geladen werden, in diesem Container ablaufen und im Speicher verbleiben bis sie wieder „entladen“ werden. Bei jedem Aufruf wird ein eigener Thread gestartet der den Aufruf abarbeitet. Das Servlet wird danach jedoch noch nicht entladen. Dadurch können Anfragen sehr rasch ausgeführt werden. Ob und wann ein Servlet wieder aus dem Speicher entfernt wird entscheidet der Servlet Container. Man spricht in diesem Zusammenhang vom *Servlet Livecycle*:

	Aus der Sicht des Benutzers	**aus der Sicht des Programms**
1	Aufruf eines Servlets (Client-Request)	-
2	der Webserver *lädt* den Servlet-Code	-
3	die Servlet-Engine *initialisiert* das Servlet	der Konstruktor und die Methode `init()` werden aufgerufen
4	die Servlet-Engine *startet* das Servlet	die Methode `service()` wird aufgerufen
4a	das Servlet läuft ab (der Benutzer sieht die Ausgabe des Servlets als HTML-codierte Seite)	servletspezifischer Code - die Service-Methode - wird ausgeführt
5	der Benutzer klickt auf einen Link und eine andere Web-Seite wird vom Browser geladen.	das Servlet bleibt im Speicher der JVM und Threads können im Hintergrund weiterlaufen
6	Erneuter Aufruf des Servlets - die JVM *startet* das Applet erneut	die Methode `service()` wird erneut aufgerufen - das Servlet wird nur erneut gestartet - **nicht** initialisiert!
7	Der Web-**Server** (die Servlet-Engine) wird heruntergefahren	Die Methode `destroy()` wird aufgerufen.

Abbildung 2-2
Servlet-Livecycle

Servlets interagieren mit dem Web-Client über Requests und Responses. Während der Client seine Anfrage über HTTP schickt, übergibt der Servlet Container die zur Bearbeitung notwendigen Daten (URI, Parameter, Session) als ServletRequest-Objekt an das Servlet. Weiters wird auch ein Response-Objekt an das Servlet übergeben, in welches das Servlet seine Antwort an den Client schreibt. Hat das Servlet die Bearbeitung der Anfrage beendet

liest der Container die Daten aus dem Response-Objekt und schickt die Daten per HTTP an den Client zurück.

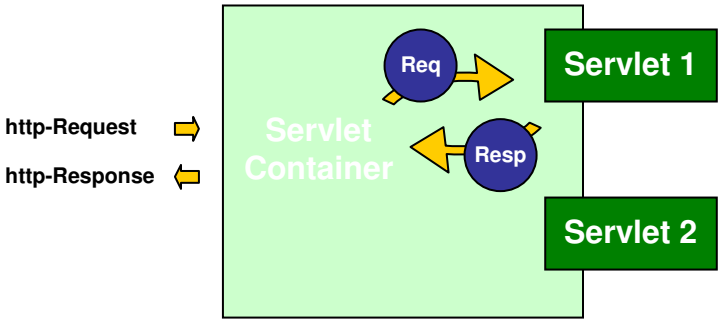

Abbildung 2-3
Servlet-Container

Das folgende Programmbeispiel illustriert den Servlet-Livecycle.

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletLivecycleDemo extends GenericServlet
{
  int i;
  int s;

  // Servlet-Construktor
  public ServletLivecycleDemo()
  {
    i=0;
    s=0;
  }

  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    i++;
  }

  public void service(ServletRequest request, ServletResponse response)
                    throws ServletException, IOException
  {
    s++;

    response.setContentType("text/html");

    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<html>");
    out.println("<head><title>ServletLivecycleDemo</title></head>");
    out.println("<body>");
    out.println("init:"+i+"<br />");
    out.println("service:"+s+"<br />");
    out.println("</body></html>");
    out.close();
  }

  public void destroy()
  {
  }
```

```
  public String getServletInfo()
  {
    return "Servlet-Livecycle-Demo";
  }
}
```

Beispiel 2-1
`ServletLivecycleDemo.java` *zur Illustration des Servlet-Livecycle*

In diesem Servlet wurden die zwei *Instanz*variablen `i` und `s` zur Speicherung der Anzahl der Aufrufe verwendet. Der Inhalt dieser Variablen bleibt bis zum Aufruf der Methode `destroy()` erhalten. Der Konstruktor und die `init()`-Methode werden nur nach dem Laden des Servlets aufgerufen. Sollten mehrere Requests an ein Servlet von verschiedenen Clients „gleichzeitig" eintreffen, so wird die `init()`-Methode trotzdem nur ein einziges Mal aufgerufen und für jeden Request ein eigener Thread des Servlets gestartet bei dem nur die `service()`-Methode aufgerufen wird. Die Methode `service()` ist eine allgemeine Methode von `javax.servlet.Servlet` bzw. `javax.servlet.http.HttpServlet`, zur Abarbeitung von Requests, die normalerweise für Web-Anwendungen nicht überschrieben wird, da von der API speziellere Methoden zur Verfügung gestellt werden.

Es sei hervorgehoben, dass jeder dieser Threads Zugriff auf dieselben Instanzvariablen hat. Dieses Beispiel eignet sich sehr gut die Anzahl der Aufrufe anzuzeigen, es sollte in der Praxis jedoch vermeiden werden veränderliche Werte in Instanzvariablen zu speichern, da diese Vorgehensweise nicht *thread-safe* ist [25]. Das bedeutet, dass man nie sicher sein kann, ob nicht ein anderer, parallel ablaufender, Thread einer zur gleichen Zeit eintreffenden Anfrage, diese Werte verändert. In der Praxis werden Instanzvariablen nur selten verwendet werden. Die Frage der Thread-Sicherheit und Synchronizität stellt sich aber im Grunde bei jedem Zugriff auf gemeinsame Ressourcen einer Webapplikation wie z.B. das ServletContext-Objekt oder Datenbank-Verbindungen. Da beim Einsatz von JSP, JSTL, Struts und JSF Objekte von diesen Techniken vielfach Gebrauch gemacht wird, soll die Problematik kurz einer näheren Betrachtung unterzogen werden:

Zeit	Thread 1	Thread 2	Inhalt:s	Ausgabe
0	…	…	0	
1	`s++;`		0 -> 1	
2		`s++;`	1 -> 2	
3		`System.out.println("start:"+s);`	2	"start:2"
4	`System.out.println("start:"+s);`		2	"start:2"

Abbildung 2-4
gegenseitige Beeinflussung zweier Threads

Soll ein Programm „thread-safe" sein, so muss man entweder

- veränderliche Werte als *lokale* Variable definieren oder
- den Zugriff darauf *synchronisieren*:

z.B.:

```
...
synchronized(this)
{
  s++;
  System.out.println("start:"+s);
  ...
  out.println("service:"+s+"<br />");
}
...
```

Beispiel 2-2
Thread-safe mittels `synchronized()`

Weitere Möglichkeiten der Synchronisation sind:

- die ganze Methode `service()` als `synchronized` zu definieren oder
- das Interface `SingleThreadModel` zu implementieren.

Diese beiden letzteren Vorgehensweisen sind allerdings mit starken Leistungseinbußen verbunden und daher zu vermeiden! Das SingleThreadModel ist darüber hinaus mit Servlet API 2.4 deprecated – d.h. es sollte nicht mehr verwendet werden.

Wie schon erwähnt ist es normalerweise nicht notwendig, dass für (HTTP-basierte) Web-Anwendungen die Methode `service()` zu überschreiben. Die Servlet API stellt die Methoden `doGet()`, `doPost()`, `doPut()`, `doHead()`, `doDelete()`, `doOptions()` und `doTrace()` in der Klasse `javax.servlet.http.HttpServlet` zur Verfügung, welche wiederum selbst von `service()` aufgerufen werden - abhängig davon welches HTTP-Kommando (`GET`, `POST`, usw.) an den Server gesendet wurde. Sie sind besser für Web-Anwendungen geeignet, da sie als Parameter Objekte vom Typ `HttpServletRequest` und `HttpServletResponse` erhalten, welche gegenüber `ServletRequest` und `ServletResponse` um zusätzliche protokollspezifische Details erweitert sind.

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Test extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                   throws ServletException, IOException
  {
    String param = request.getParameter("test");

    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());

    out.println("<html>");
    out.println("<head><title>Test</title></head>");
    out.println("<body>");

    if ((param==null) || (param.equals("")))
      out.println("no parameter 'test' or no value");
    else
      out.println("test="+param);

    out.println("</body></html>");
```

```
    out.flush();
    out.close();
  }
}
```

Beispiel 2-3
HttpServlet-Beispiel `Test.java`

Dieses Servlet überschreibt nur die `doGet()`-Methode und erfüllt eine einfache Aufgabe: es überprüft ob im von der Servlet-Engine übergebenen `ServletRequest`-Objekt ein vom Client übersendeter Parameter namens „Test“ existiert. Danach wird dem `Response`-Objekt zuerst der MIME-Type der Antwort mitgeteilt und dann der `OutputStream` erfragt. In diesen wird dann die gesamte Ausgabe geschrieben: Ist der Parameter test nicht vorhanden oder leer sein wird eine Meldung erzeugt, dass kein Parameter übergeben wurde. Andernfalls wird der Wert des Parameters ausgegeben.

Um dieses Servlet verwenden zu können muss es der Servlet-Engine bekannt gemacht werden. Die geschieht, wie schon erwähnt, in `web.xml`:

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>at.ac.akhwien.mvc.servlets.Test</servlet-class>
  </servlet>

<!-- http://servername:8080/servlets/servlet/test -->
    <servlet-mapping>
        <servlet-name>
            test
        </servlet-name>
        <url-pattern>
            /servlet/test
        </url-pattern>
    </servlet-mapping>

</web-app>
```

Beispiel 2-4
Definition des Servlet-Aufrufs in web.xml

Unter der Annahme, dass die Web-Applikation im Verzeichnis `servlets` liegt, erfolgt der Aufruf im Web-Browser unter der Adresse
`http://servername:8080/servlets/servlet/test`.

Schon an diesem einfachen Beispiel wird deutlich, wo das Problem bei der Benutzung von Servlets ist: Code und Ausgabe sind untrennbar durchmischt!

Eine Möglichkeit wie HTML-Code aus HTML-Dateien oder die Ausgabe anderer Servlets in ein Servlet mit den Bordmitteln der Servlet-API eingebunden werden kann (include) oder die Ausgabe eines Servlets auf ein anderes Servlet weitergeleitet werden kann (forward), ist die

Verwendung des `RequestDispatcher`-Objektes. JavaServer Pages, Struts und JSF machen vom `RequestDispatcher`-Mechanismus regen Gebrauch.

RequestDispatcher

Mit der Servlet-API 2.1 wurde das `RequestDispatcher`-Objekt eingeführt. RequestDispatcher dienen zum Einfügen von oder Umleiten der Ausgabe an andere Servlets. Das `RequestDispatcher`-Objekt hat nur zwei Methoden: `forward()` und `include()`.

Die folgenden Beispiel-Servlets implementieren die Funktionalität des vorangegangenen Beispiels unter der Verwendung der `forward()` und der `include()`-Funktion. Sie überprüfen ob ein Parameter namens test mit einem Wert übergeben wurde oder nicht. Dementsprechend wird eine Meldung zurückgeliefert.

Forward

Forwarding funktioniert innerhalb des Servers und ist somit für den Benutzer nicht zu bemerken. Der Forward-Mechanismus sollte nicht mit einem Redirect verwechselt werden. Bei einem Redirect erhält der Client (Web-Browser) vom Server den HTTP-Statuscode MOVED PERMANENTLY und den Header „Location“ mit der neuen URL übermittelt, die dann der Client selbst selbständig abfragt. Die Servlet API enthält dazu die Methode `sendRedirect()` in `HttpServletResponse`.

Im nächsten Beispiel verzweigt das Servlet auf eine HTML-Seite. Die Seite `input.html` ruft ein Servlet entweder mit oder ohne Parameter `test` auf. Wird das Servlet ohne Parameter aufgerufen, erfolgt ein Forward zurück auf `input.html`. Wird ein Parameter an das Servlet übergeben, wird auf die Seite `output.html` weitergeleitet. Der RequestDispatcher wird hier mittels der Methode `getRequestDispatcher()` des `ServletContext`-Objekts angefordert und erfordert eine absolute Pfadangabe. Seit Servlet-API 2.2 enthält auch die Klasse `ServletRequest` eine `getRequestDispatcher()`-Methode, der man auch einen relativen Pfad übergeben kann.

```
<html>
  <head>
    <title>Input - forward()</title>
  </head>
  <body>
    RequestDispatcher - forward()<br />
    <ul>
      <li><a href="/servlets/servlet/forward">forward</a></li>
      <li><a href="/servlets/servlet/forward?test=qwerty">
          forward?test=qwerty</a></li>
    </ul>
  <body>
</html>
```

Beispiel 2-5
`input.html`

```
<html>
  <head>
    <title>Output</title>
  </head>
```

```
  <body>
    Parameter test wurde übergeben! <br />
    Parameter test was given!<br />
    <hr />
    Zur&uuml;ck zum <a href="/servlets/input.html">Start</a>
  <body>
</html>
```

Beispiel 2-6
`output.html`

```
package At.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class forward extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                   throws ServletException, IOException
  {
    String param = request.getParameter("test");

    ServletContext context = getServletContext();


    if ((param==null) || (param.equals("")))
    {
      RequestDispatcher rd = context.getRequestDispatcher("/input.html");
      rd.forward (request,response);
    }
    else
    {
      RequestDispatcher rd = context.getRequestDispatcher("/output.html");
      rd.forward (request,response);
    }
  }
}
```

Beispiel 2-7
RequestDispatcher-Servlet `forward.java`

Der Nachteil an diesem Beispiel ist, dass an die HTML-Seiten keine Werte zur Ausgabe übergeben werden können. Auf diese Art und Weise kann weder der Parameter auf einer Output-Seite zur Bestätigung ausgegeben werden, noch eine fehlerhafte Eingabe zur Korrektur an eine Input-Seite zurückgeschickt werden.
Wie dies bewerkstelligt werden kann, soll am Beispiel der Bestätigungsmeldung durch die Output-Seite gezeigt werden.

Include

Nun werden 2 (halbe) HTML-Seiten bei der Ausgabe inkludiert: `outputheader.html` und `outputfooter.html`. Zwischen ihren Aufrufen wird der Parameter ausgegeben.

```
<html>
  <head>
    <title>Output</title>
  </head>
  <body>
```

Beispiel 2-8
`outputheader.html`

```
    <hr />
    Zur&uuml;ck zum <a href="/servlets/input.html">Start</a>
  <body>
</html>
```

Beispiel 2-9
`outputfooter.html`

```
package At.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class include extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
  {
    String param = request.getParameter("test");

    ServletContext context = getServletContext();
    RequestDispatcher rd;

    if ((param==null) || (param.equals("")))
    {
      rd = context.getRequestDispatcher("/input.html");
      rd.forward (request,response);
    }
    else
    {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      rd = context.getRequestDispatcher("/outputheader.html");
      rd.include (request,response);

      out.println("  Parameter test="+param+" wurde übergeben! <br />");

      rd = context.getRequestDispatcher("/outputfooter.html");
      rd.include (request,response);

    }
  }
}
```

Beispiel 2-10
RequestDispatcher-Servlet `include.java`

In diesem Beispiel ist aber erneut Programm-Logik und Ausgabe vermischt! Der Lösungsansatz besteht darin, dieses Beispiel um ein zusätzliches Servlet zu erweitern.

Während ein Servlet als „Controller“ fungiert und sich um die if-Abfrage und Weiterverzweigung (forward) kümmert, sorgt das „View-Serlvet“ für die Ausgabe der einzelnen Komponenten (include). Diese Vorgehensweise entspricht dem oben schon erwähnten Model-View-Controller-Konzept - „Model 2“.

Es muss dafür gesorgt werden, dass das zweite Servlet die Daten erhält die es anzeigen soll. Die Parameter, die das erste Servlet übergeben bekommen hat, können durch den RequestDispatcher einfach weitergereicht werden. Da das Controller-Servlet auch für Zugriffe auf das Model zuständig ist, müssen unter Umständen auch größere Datenstrukturen oder Binär-Daten zur Darstellung an das View-Servlet weitergereicht werden, die nicht in Parameterform als Zahlen und Strings darstellbar sind.

Wie bei der Einführung in die Funktionsweise von Servlets und Servlet-Engine oben schon beschrieben, handelt es sich bei den dem Servlet übergebenen Request-Daten um ein Objekt. Dieses `ServletRequest`-Objekt enthält auch Methoden mittels derer andere Objekte (z.B. JavaBeans) im ServletRequest-Objekt unter einem Namen hinterlegt bzw. wieder abgerufen werden können. Die Methode `setAttribute()` dient zum Hinterlegen, die Methode `getAttribute()` zum Abrufen des Objektes aus dem `ServletRequest`-Objekt. Weiters existieren die Methoden `getAttributeNames()` zum Auflisten der hinterlegten Objekte und `removeAttribute()` zum Löschen von Objekten.

In Analogie zum oben vorgestellten Model 2 der JSP kann man sich die Funktionsweise nun so vorstellen:

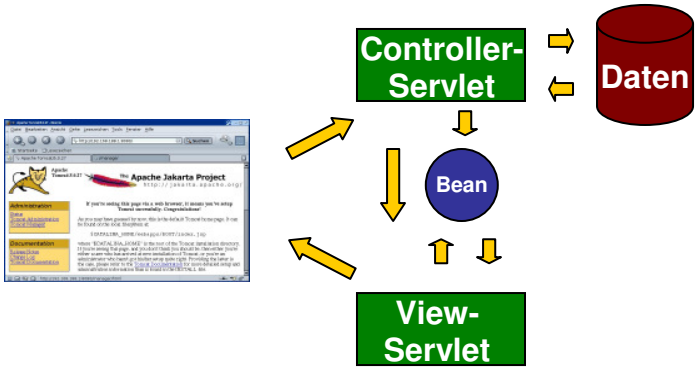

Abbildung 2-5
Model 2 mit Servlets

Im folgenden Beispiel werden beide Möglichkeiten der Datenübergabe gezeigt: sowohl über Parameter, als auch über Objekte. Der Parameter der dem Controller-Servlet übergeben wird, wird weitergereicht und zusätzlich wird ein weiterer Parameter übergeben. Mit Hilfe von `setAttribute()` / `getAttribute()` erhält das `HttpServletRequest-Objekt` ein `String`-Objekt.

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class controller extends HttpServlet
{
```

```
  public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
  {
    String param = request.getParameter("test");

    ServletContext context = getServletContext();
    RequestDispatcher rd;

    if ((param==null) || (param.equals("")))
    {
      rd = context.getRequestDispatcher("/input.html");
      rd.forward (request,response);
    }
    else
    {
      request.setAttribute("nachname", "Kornfeil");

      rd = context.getRequestDispatcher("/servlet/view?vorname=Harald");
      rd.forward (request,response);

    }
  }
}
```

Beispiel 2-11
RequestDispatcher-Servlet `controller.java`

Das View-Servlet gibt alle im `ServletRequest`-Objekt erhaltenen Parameter und Attribute aus:

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class view extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
  {
    Enumeration aufzaehlung = null;
    String name;
    String wert;
    String attr = null;
    RequestDispatcher rd;

    response.setContentType("text/html");

    ServletContext context = getServletContext();

    rd = context.getRequestDispatcher("/viewheader.html");
    rd.include (request,response);

    PrintWriter out = response.getWriter();

    out.println("Parameter:\n <ol> ");
    aufzaehlung = request.getParameterNames();
    for(;aufzaehlung.hasMoreElements();)
    {
      name = (String) aufzaehlung.nextElement();
      wert = request.getParameter( name);
      out.println("<li>"+name+": "+wert+"</li>\n");
    }
```

```
    out.println("</ol> ");

    out.println("Attribute:\n <ol> ");
    aufzaehlung = request.getAttributeNames();
    for(;aufzaehlung.hasMoreElements();)
    {
      name = (String) aufzaehlung.nextElement();
      out.println("<li>"+name+"</li>\n");
    }
    out.println("</ol> ");

    attr = (String) request.getAttribute("nachname");
    out.println("    <i>&copy; 2006 by "+request.getParameter("vorname")+
                " "+attr+"</i>");

    rd = context.getRequestDispatcher("/viewfooter.html");
    rd.include (request,response);
  }
}
```

Beispiel 2-12
RequestDispatcher-Servlet `view.java`

```
<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
```

Beispiel 2-13
HTML-Header-Datei `viewheader.html`

```
  </body>
</html>
```

Beispiel 2-14
HTML-Footer-Datei `viewfooter.html`

Dieser Lösungsansatz nach Model 2 mag etwas kompliziert erscheinen, bietet jedoch eine sehr elegante Art der Trennung von Ausgabe und Logik. Die Trennung ist zwar noch nicht perfekt, doch wie später noch gezeigt wird, kann das Model 2 mit JavaServer Pages realisiert werden, da für JSPs sowohl spezielle Zugriffs-Funktionen auf Objekte als auch Forward- und Include-Mechanismen zur Verfügung stehen, die es dem Programmierer sehr viel einfacher machen. Auch das nach dem Model 2-Konzept entwickelte Struts-Framework funktioniert auf Basis dieser Grundtechniken.
Neben `(Http)ServletRequest` bieten auch die Objekte `ServletContext` und `HttpSession` die Möglichkeit, dass über diese beiden Methoden beliebige Objekte in ihnen abgelegt werden können. Diese Objekte unterscheiden sich in Bezug auf hinterlegte Objekte vor allem durch ihre unterschiedliche Gültigkeitsdauer bzw. Gültigkeitsbereiche – so genannte *Scopes*.
Ein `(Http)ServletRequest`-Objekt existiert nur für die aktuelle Anfrage und nur an der Anfrage beteiligte Servlets haben darauf Zugriff.
Das `ServletContext`-Objekt wird beim Start der Web-Applikation erstellt und alle Servlets der Web-Applikation können es nutzen.
Das `HttpSession`-Objekt besteht für die Dauer einer Session. Alle an der Session beteiligten Servlets können darauf zugreifen.

Sitzungsmanagement

Eine Sitzung könnte man definieren als eine Menge von Daten, die über den Status in dem sich der Benutzer innerhalb der Web-Anwendung befindet Auskunft geben. Dies könnten z.B. der Inhalt eines Warenkorbes einer e-Commerce-Anwendung oder die Identifikationsdaten des Benutzers eines Web-Mail-Systems sein.

Da es sich bei HTTP um ein zustandsloses Protokoll handelt, hat ein Server bzw. eine Web-Anwendung vom Protokoll her keine Möglichkeit festzustellen ob verschiedene Anfragen einer Adresse auch von ein und demselben Client kommen. Um sicherzustellen, dass Daten dem Client zugeordnet werden, der eine Sitzung begonnen hat, muss der Programmierer eine von HTTP unabhängige Sitzungsverfolgung realisieren.

Es stehen grundsätzlich verschiedene Möglichkeiten des Sitzungsmanagements [26] [27] zur Verfügung. Sie bieten verschiedene Besonderheiten, die in manchen Fällen einen Vorteil, in anderen Fällen einen Nachteil darstellen können:

- „Basic-Authentifizierung“ durch den WebServer
  Der Server erkennt anhand des URL-Pfades ob eine Authentifizierung notwendig ist. Er fordert den Client zur Authentifizierung mit Username und Passwort auf. Der Client merkt sich diese Daten und schickt sie bei jeder weiteren Anfrage an eine Ressource dieses Servers wieder mit. Den Benutzernamen erhält man mit der Methode `getRemoteUser()` des Objekts `HttpServletRequest`
  Die Basic-Authentification ist kodiert (BASE64) jedoch nicht verschlüsselt. Sie bietet daher keine Sicherheit für sensible Daten.
  Besonderheit: Damit die Daten nicht mehr mit gesendet werden, muss der Browser meist beendet und neu gestartet werden.
- HIDDEN-Fields in Formularen
  In Formularen können z.B. mittels `<input type="hidden" name="username" value="harald" />` Daten hinterlegt werden, die der Browser nicht sichtbar rendert, aber beim Absenden des Formulars als Parameter an den Server mit der Anfrage mitgeschickt werden.
  Besonderheit: Die Daten sind im Quelltext der Seite lesbar. Wenn die Web-Seite abgespeichert wird, werden daher die Daten ebenfalls mitgespeichert. Eine Sitzung kann so gespeichert und eventuell später fortgesetzt werden. Jeder der Zugriff auf die gespeicherte Seite hat kann die Sitzung fortsetzen.
- URL-Rewriting
  Der Server schreibt bei jeder von ihm abgelieferten Seite automatisch die URLs der darin enthaltenen Links in der Form um, dass sie danach eine eindeutige Kodierung enthält. Dies kann durch eine zusätzliche Pfadangabe, angehängte Parameter oder durch z.B. zusätzlich angehängte Zeichenketten nach einem Strichpunkt geschehen. Durch die Eindeutigkeit dieser Kodierung, welche bei der nächsten Anfrage an den Server wieder mitgeschickt wird, ist eindeutig sichergestellt, von welchem Client die Anfrage kommt.
  Die Servlet-API stellt in `HttpServletResponse` seit Version 2.1 dazu die Methoden `encodeURL()` und `encodeRedirectURL()` zur Verfügung die Methoden sind `encodeUrl()` und `encodeRedirectUrl()` sind als `deprecated` gekennzeichnet und sollten nicht mehr verwendet werden.
  Besonderheit: der URL inklusive Kodierung kann als Bookmark abgespeichert werden. Eine Sitzung kann so gespeichert und eventuell später fortgesetzt werden. Jeder der diese URL kennt, kann die Sitzung fortsetzen.

- Cookies
  Ein Cookie dient zum Speichern von Informationen auf dem Client. Der Server sendet auf eine Client-Anfrage in seinem Antwort-Header eine Aufforderung zum Speichern eines Cookies mit. Dieses Cookie kann folgende Daten enthalten: ein oder mehreren Name/Wert-Paare, eine Domain und eine Pfadangabe innerhalb der das Cookie gültig ist bzw. abgefragt werden darf, ein Ablaufdatum der Daten, die Information, dass das Cookie nur über eine HTTPS-Verbindung gesendet werden darf, und einige andere Daten mehr.
  Anhand der übertragenen Cookie-Daten lässt sich ein Client identifizieren.
  Besonderheit: Cookies lassen sich von modernen Browsern sperren. Man kann nicht mit Sicherheit davon ausgehen, dass ein Client Cookies akzeptiert und dass eine Verwendung von Cookies möglich ist. Weiters muss man sich bewusst sein, dass am Client noch vorhandene Cookies einer alten Sitzung, die noch nicht abgelaufen sind, vom Client übertragen werden und durch ihre ungültige Daten Verwirrung stiften können.
- SSL/TLS:
  Der Secure Sockets Layer und dessen Weiterentwicklung Transport-Layer-Security sind HTTP-Erweiterungen zur Verschlüsselung von Daten nach dem Prinzip der Public-Key-Kryptographie und stellen ein eigenes Sitzungskonzept zur Verfügung.
  Besonderheit: man benötigt dazu aktuelle digital signierte Zertifikate am Server und am Client.

JavaServlets und JSPs unterstüzten alle obigen Lösungsansätze. Um den Umgang mit Sitzungen zur vereinfachen können Servlets und JSPs das `HttpSession-Objekt nutzen` [28].

`HttpSession`-Objekte werden im Server unter einer Id gespeichert, und diese Id wird entweder mittels URL-Rewriting oder Cookies an den Client gesendet. Dies hat den Vorteil, dass sensible Daten nicht bei jeder Client-Anfrage und Server-Antwort hin- und her übertragen werden müssen. Weiters können wie schon erwähnt innerhalb eines `HttpSession`-Objekts wieder beliebige Java-Objekte abgespeichert werden.

Mit der Servlet-Spezifikation 2.2 haben sich die Methoden mit denen auf das `HttpSession`-Objekt schreibend und lesend zugegriffen werden kann geändert. Statt der Methoden `putValue()`, `getValue()`, `getValueNames()` und `removeValue()` stehen jetzt wie beim `(Http)ServletRequest`-Objekt und `ServletContext`-Objekt die Methoden `setAttribute()`, `getAttribute()`, `getAttributeNames()` und `removeAttribute()` zur Verfügung, um den Zugriff einheitlich zu gestalten.

Mittels des Aufrufes der Methode `getSession(true)` können `HttpSession`-Objekte in Servlets neu erzeugt werden bzw. es kann ein Servlet an einer schon bestehenden Sitzung teilnehmen. Ob eine JSP an einer Sitzung teilnehmen soll wird über die `page`-Direktive mit dem Attribut `session` festgelegt - siehe dazu später.

Die folgende Web-Anwendung illustriert den Einsatz von Sessions im Zusammenspiel mit einem Controller-Servlet und einem View-Servlet unter der Verwendung von Cookies. Die Aufgabe der Anwendung ist es Daten, die über ein `<input />` -Feld eingegeben werden, in einem Session-Objekt zu speichern und anzuzeigen. Die Session muss über den Start-Link gestartet werden um Daten speichern zu können. Erst dann speichert die Anwendung die über das Eingabefeld übermittelten Daten. Die Session wird mit dem Ende-Link beendet, wodurch die im Session-Objekt gespeicherten Daten verloren gehen.

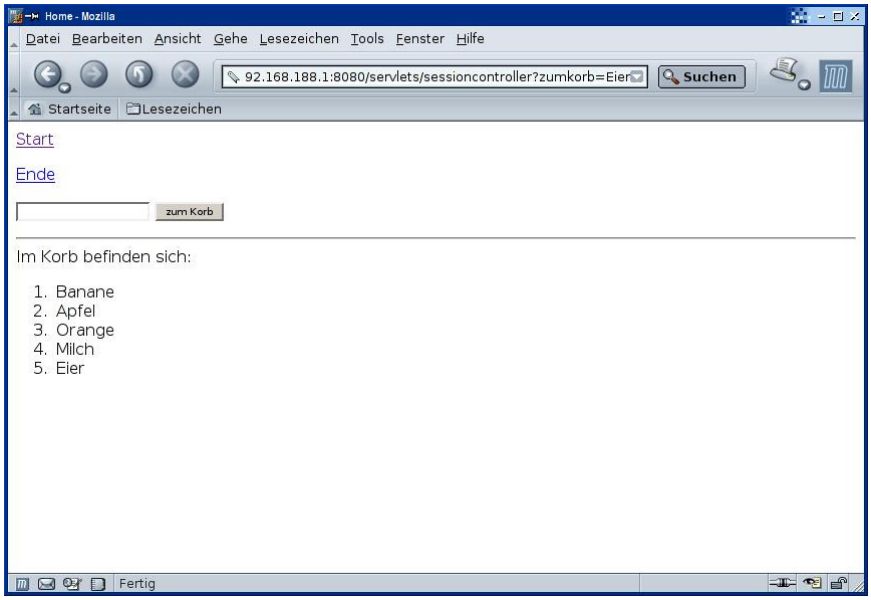

Abbildung 2-6
Auflistung der im Session-Objekt gespeicherten Daten

Zuerst überprüft das Controller-Servlet ob sich die Anwendung schon in einer Sitzung befindet. Besteht noch keine Session und es soll eine gestartet werden, dann wird ein Session-Objekt erzeugt und es erfolgt ein Forward auf das View-Servlet – andernfalls wird zur Eingabeseite zurück weitergeleitet.
Besteht schon eine Sitzung, wird überprüft ob sie beendet (ungültig gesetzt) werden soll. Wenn ja, dann wird wieder auf die Eingabeseite weitergeleitet. Soll die Sitzung nicht beendet werden, wird ein eventuell übergebener Parameter im Session-Objekt gespeichert und es erfolgt die Anzeige über das View-Servlet.

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class sessioncontroller extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
  {
    String            wert = null;
    HttpSession       session = null;
    RequestDispatcher rd = null;
    Vector            liste = null;


    wert=request.getParameter("todo");

    session=request.getSession(false);

    if (session==null)
    {

      if ((wert!=null) && (wert.equals("start")))
      {
        session=request.getSession(true);
        rd = request.getRequestDispatcher("sessionview");
```

```
        rd.forward (request,response);
        return;
      }
      else
      {
        rd = request.getRequestDispatcher("index.html");
        rd.forward (request,response);
        return;
      }
    }

    if ( (wert!=null) && (wert.equals("ende")) )
    {
      session.invalidate();
      rd = request.getRequestDispatcher("index.html");
      rd.forward (request,response);
      return;
    }

    wert=request.getParameter("zumkorb");

    if (wert!=null)
    {
      liste= (Vector) session.getAttribute("Korb");

      if (liste==null)
      {
        liste = new Vector();
      }
      liste.add(wert);

      session.setAttribute("Korb",liste);
    }

    rd = request.getRequestDispatcher("sessionview");
    rd.forward (request,response);

  }
}
```

Beispiel 2-15
Session-Controller-Servlet `sessioncontroller.java`

Auch das View-Servlet überprüft zuerst, ob schon eine Sitzung besteht. Wenn nicht, findet eine Weiterleitung zur Anfangsseite statt. Erfolgt keine Weiterleitung kommt es zur Darstellung: es werden zuerst eine Datei mit HTML-Headern und eine Datei die der Eingabeseite entspricht (ohne HTML-Header und -Footer) eingebunden. Danach erfolgt die Listendarstellung der im Session-Objekt gespeicherten Daten. Zuletzt erfolgt noch ein Include einer Datei mit dem HTML-Footer.

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class sessionview extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
  {
    String            wert = null;
```

```
    HttpSession       session = null;
    RequestDispatcher rd = null;
    Vector            liste = null;

    session = request.getSession(false);
    if (session==null)
    {
      rd = request.getRequestDispatcher("index.html");
      rd.forward (request,response);
      return;
    }

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    rd = request.getRequestDispatcher("viewheader.html");
    rd.include (request,response);

    rd = request.getRequestDispatcher("sessionview.html");
    rd.include (request,response);

    out.println("Im Korb befinden sich:\n <ol> ");

    liste= (Vector) session.getAttribute("Korb");
    if (liste!=null)
    {
      for (int i=0;i<liste.size();i++)
      out.println("<li>"+liste.get(i)+"</li>\n");
    }
    out.println("</ol> ");

    rd = request.getRequestDispatcher("viewfooter.html");
    rd.include (request,response);

  }
}
```

Beispiel 2-16
Session-View-Servlet `sessionview.java`

Der vollständige Code der Web-Anwendung befindet sich im Anhang.

Nach der Ausführlichen Besprechung der grundlegenden Programmiertechniken auf Basis der Servlet-API soll nun näher auf die Funktionsweise und Programmierung von JavaServer Pages eingegangen werden, da JSTL, Struts und auch JavaServer Faces auf JavaServer Pages-Techniken aufbauen, ist das Verständnis über deren Funktionsweise Voraussetzung für deren Verständnis. Daher wird die Programmierung von JSPs etwas ausführlicher vorgestellt.

2.4. JavaServer Pages – JSP [29] [30] [31]

Allgemeines

Eine JSP entspricht im Grunde einer HTML-Datei, die spezieller Tags enthält. Diese Tags dienen zur Ausgabe und Verarbeitung von Daten sowie zur Ablaufsteuerung.

JSP-Anweisungen werden in `<%` ... `%>` eingeschlossen, wobei man *Direktiven* („*directives*“), *Skript-Tags* („*scripting elements*“) *Aktionen* („*actions*“) und *Kommentare* („*comments*“) unterscheidet.

- *Direktiven* erzeugen keinerlei Ausgaben, sondern senden Nachrichten an den JSP-Container. Durch das Setzen gewisser Parameter kann die Übersetzung der JSPs in

Servlets beeinflusst werden (Einbinden von Tag-Libraries, Teilnahme an Sessions, usw.)
- *Skript-Tags* enthalten Programmlogik in Form von Quellcodefragmenten.
- *Aktionen* ähneln von ihrer Syntax her Tag-Libraries und können als eine Art Standard-Tag-Library gesehen werden, die eine Mehrfachverwendung von Web-Komponenten erleichtern soll. Aktionen bieten z.B. einen einfacheren Umgang mit JavaBeans und eigene Tags für das Weiterleiten auf und Einfügen von anderen HTML-Seiten und JSPs.
- *Kommentare* dienen zur Kommentierung von JSPs und erscheinen nicht in der Ausgabe einer JSP.

Wird von einem Client eine JSP aufgerufen, wird von der Servlet-Engine folgender Ablauf in Gang gesetzt:
- es wird überprüft ob die JSP schon einmal übersetzt wurde,
- wenn nicht wird die JSP-Datei eingelesen,
- die JSP-Datei wird in ein JavaServlet umgewandelt: der statische HTML-Teil wird über `println()`-Befehle ausgegeben und die Programmlogik wird an den entsprechenden Stellen in den Code integriert,
- der so entstandene Quell-Code wird in Java-Byte-Code übersetzt,
- das resultierende Servlet wird ausgeführt,
- bei jedem weiteren Aufruf der JSP wird automatisch das Servlet, welches schon übersetzt ist und vom Server geladen wurde, erneut ausgeführt.

Dazu ein Beispiel anhand einer HTML-Seite ohne JSP-Tags:

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Beispiel 2-17
Beispiel-JSP `helloworld.jsp`

Wird diese JSP nun von einem Browser aufgerufen, wird der HTML-Code in folgenden Java-Servlet-Code umgewandelt:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
```

```
public class _0002fhelloworld_0002ejsphelloworld_jsp_0 extends HttpJspBase {


    static {
    }
    public _0002fhelloworld_0002ejsphelloworld_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse  response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                  "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="E:\\jakarta-tomcat-3.2.1\\webapps\\servlets-
stepbystep\\helloworld.jsp";from=(0,0);to=(7,7)]
                out.write("<html>\r\n  <head>\r\n    <title>Hello
World</title>\r\n  </head>\r\n  <body>\r\n    <h1>Hello World</h1>\r\n
</body>\r\n</html>");
            // end

        } catch (Exception ex) {
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

Beispiel 2-18
Servlet-Code umgewandelte Beispiel-JSP **`helloworld.jsp`**

Es ist deutlich zu sehen, dass der Servlet-Quellcode viele Codeteile enthält, die eigentlich nicht benötigt werden. Diese werden zwar vom Java-Compiler teilweise herausoptimiert, jedoch bleibt ein gewisser Overhead erhalten, wodurch eine JSP niemals so schnell ausgeführt werden kann wie ein speziell programmiertes Servlet.

Skripte

Bei Skripten unterscheidet man

- *Anweisungen* – engl.: *scriptlets*
- *Ausdrücke* – engl.: *expressions*
- *Deklarationen* – engl.: *declarations*

Anweisungen – Scriptlets

Scriptlets sind kleine Java-Programmfragmente die in die innerhalb spezieller Tags zur Programmablaufsteuerung eingesetzt werden. Dieser Programmcode entspricht dem Code den Servlets in der Servlet-Methode `service()` bzw. in den `HttpServlet`-Methoden `doget()`, `dopost()` usw. enthalten. Alle innerhalb von Scriptlets definierten Variablen sind nur als *lokale* Variablen definiert (auch *Instanz*variablen können definiert werden - siehe „Vereinbarungen“ unten).

Um die Ausgabe von HTML-Code zu vereinfachen kann in einem mit geschwungenen Klammern eingeschlossener Programmcode-Abschnitt das Scriptlet vorübergehend beendet werden. Darauf folgt der HTML-Code und danach muss die Klammer selbstverständlich wieder innerhalb eines Scriptlet-Tags geschlossen werden. Der dazwischen liegende HTML-Code wird vom JSP-Container vor der Übersetzung in ein Servlet in out.println()-Anweisungen umgewandelt. Das folgende Beispiel zeigt die Verwendung von Scriptlet-Tags und die Möglichkeit der Einbettung von HTML-Code: es wird eine lokale Variable i initialisiert und ihr wert in Worten angezeigt.

```
<html>
  <head>
    <title>JSP ( Anweisungen )</title>
  </head>
  <body>
    <% int i = 0; %>
    Der Wert von i ist:
    <%
      if (i == 0)
      {
        %> <b>Null</b> <%
      }
      else
      {
        %> ungleich Null <%
      }
    %>
  </body>
</html>
```

Beispiel 2-19
Beispiel-JSP **`scriptlet.jsp`**

Sollte ein Fehler in einem Scriptlet eine Exception erzeugen so kann diese natürlich über den üblichen try-catch-Mechanismus von Java abgefangen werden. JSPs stellen aber auch eine andere Möglichkeit zur Verfügung - siehe dazu später.

Um die Programmierung von Scriptlets zu erleichtern, sind standardmäßig einige oft benützte Servlet-Objekte als „implizite JSP-Objekte“ („implicit JSP-objects“) vorinstanziert. Dadurch wird es JSP-Autoren erleichtert auf diese Objekte zuzugreifen.

Wie schon bei der Servlet-Programmierung besprochen, können Objekte in verschieden Objekten, die von der Servlet-Engine bereitgestellt werden, eingebettet werden (`(Http)ServletRequest`, `ServletContext`, `HttpSession`). JSPs besitzen darüber hinaus noch das `page`–Objekt, das nur für die jeweilige JSP gilt. Auch die impliziten JSP-Objekte haben gewisse Gültigkeitsbereiche (*scopes*) innerhalb welcher auf sie zugegriffen werden kann. Genaueres dazu später bei der ausführlichen Besprechung von JavaBeans im Zusammenhang mit JSP-Aktionen.

Objektname	**Beschreibung (Klasse / Interface)**	**Scope**
`request`	Anfrage-Objekt, entspricht dem `request`-Parameter `von service()` bzw. `doget()`, `dopost()` (protokollabhängiger Untertyp von `javax.servlet.ServletRequest` z.B. `javax.servlet.http.HttpServletRequest`)	`request`
`response`	Antwort-Objekt, entspricht dem `response`-Parameter `von service()` bzw. `doget()`, `dopost()` (protokollabhängiger Untertyp von `javax.servlet.ServletResponse` z.B. `javax.servlet.http.HttpServlet.Response`)	`page`
`pageContext`	Context-Object der JSP (`javax.servlet.jsp.PageContext`)	`page`
`session`	Session-Objekt - gilt nur für HTTP-Protokoll, entspricht dem Aufruf `HttpSession session=request.getSession(true);` (`javax.servlet.http.HttpSession`)	`session`
`application`	Context-Object der Web-Applikation, entspricht dem Aufruf `ServletContext application=getServletConfig.getContext();` (`javax.servlet.ServletContext`)	`application`
`Out`	Objekt ermöglich das Schreiben in den Output-Stream (`javax.servlet.jsp.JspWriter`)	`page`
`config`	ServletConfig-Objekt der JSP, entspricht dem Aufruf `ServletConfig config=getServletConfig();` (`javax.servlet.ServletConfig`)	`page`
`Page`	= `this` (implementiert `javax.servlet.jsp.JspPage`)	`page`
`exception`	Das noch nicht abgefangene Throwable-Objekt, das aufgrund eines Fehlers „geworfen“ wurde existiert nur in Fehlerseiten (`java.lang.Throwable`)	`page`

Tabelle 2-1
Implizite JSP-Objekte

Durch Verwendung dieser Objekte gestaltet sich der Zugriff auf die API-Methoden und Eigenschaften dieser Objekte sehr einfach: sie werden durch einen Punkt getrennt aufgerufen.

```
<html>
  <head>
    <title>JSP ( implizite Objekte )</title>
  </head>
  <body>
    <%
```

```
      response.setContentType("text/plain");
      if ((request.getParameterNames().hasMoreElements())
           || (config.getInitParameterNames().hasMoreElements()))

      {
        out.println("got Parameters / Parameter erhalten");
      }
      else
      {
        out.println("no Parameters / keine Parameter");
        application.log("no parameters / keine Parameter");
      }
      session.invalidate();
    %>
  </body>
</html>
```

Beispiel 2-20
impobj.jsp

Ausdrücke – Expressions

Um die Inhalte von Variablen und Objekten ausgeben zu können, ohne Scriptlet-Code verwenden zu müssen, gibt es Ausdrücke. Bei Ausdrücken wird die Variable in ein Tag eingeschlossenen: `<%=` *Variable* `%>`. Dies entspricht dem Scriptlet `<% out.println(` *Variable* `); %>`.

```
<html>
  <head>
    <title>JSP ( Ausdruck )</title>
  </head>
  <body>
    <% int i = 0;   %>
    Der Wert von i ist: <%= i %>
  </body>
</html>
```

Beispiel 2-21
expression.jsp

Vereinbarungen – declarations

Innerhalb einer JSP Es können auch Funktionen definiert werden. Dazu schließt man sie in `<%! ... %>` ein. Hierdurch können auch *Instanz*variablen definiert werden. In Scriptlets definierte Variablen sind lediglich *lokale* Variablen der Methode _jspService.

```
<html>
  <head>
    <title>JSP ( Vereinbarungen )</title>
  </head>
  <body>
    <%! int i = 2; %>
    <%! int quad(int num)
        {
          return num*num;
        }
     %>
```

```
    <%  int i = 3; %>

    <p>
    Der Wert der <b>Instanz</b>-variable i ist: <%= this.i %>  <br />
    Der Wert der <b>Instanz</b>-variable i zum Quadrat ist <%= quad(this.i) %>
    </p>
    <p>
    Der Wert der <b>lokalen</b> Variable i ist: <%= i %>  <br />
    Der Wert der <b>lokalen</b> Variable i zum Quadrat ist <%= quad(i) %>
    </p>

  </body>
</html>
```

Beispiel 2-22
`declarations.jsp`

Auch JSPs haben eine `Init()`- und eine `Destroy()`-Methode. Ihr Verhalten bzw. der Aufruf dieser Methoden durch die JSP-Engine entspricht dem (oben beschriebenen) Aufruf bei Servlets. Sie heißen hier `jspInit()` und `jspDestroy()`.

Durch Definition innerhalb einer Vereinbarung können diese beiden Methoden überschrieben werden. Im Gegensatz zu Scriptlets ist hierbei zu beachten, dass Ausnahmen mittels `try-catch`-Mechanismus abgefangen werden müssen, da die Methoden keine Exceptions erzeugen können. Nur unchecked Exceptions dürfen geworfen werden (RuntimeException oder Error und deren Unterklassen).

```
<%! int i=0; %>
<%! // !! Ausnahme-Behandlung innerhalb der Methode zwingend !!
    // !!       Methode kann keine Exceptions erzeugen       !!
    // nur unchecked Exceptions sind moeglich:
    // d.h. RuntimeException oder Error und deren Unterklassen
    public void jspInit()
    {
      try
      {
        System.out.println("jspInit (i:"+i+")");
      }
      catch (Exception e)
      {
        throw new RuntimeException("Init-Fehler!");
      }
    }

    // !! Ausnahme-Behandlung innerhalb der Methode zwingend !!
    // !!       Methode kann keine Exceptions erzeugen       !!
    // nur unchecked Exceptions sind moeglich:
    // d.h. RuntimeException oder Error und deren Unterklassen
    public void jspDestroy()
    {
      try
      {
        System.out.println("jspDestroy (i:"+i+")");
      }
      catch (Exception e)
      {
        throw new RuntimeException("Destroy-Fehler!");
      }
    }
%>

<html>
```

```
  <head>
    <title>JSP ( spezielle vereinbarungen )</title>
  </head>
  <body>

    <% this.i = this.i+1; %>

    Der Wert der <b>Instanz</b>-variable i ist: <%= this.i %>  <br />

  </body>
</html>
```

Beispiel 2-23
`initdestroy.jsp`

Kommentare – comments

Kommentare können in Tags der Form <%-- Kommentar --%> verfasst werden und werden nicht an den Client geliefert.
Dies entspricht den Scriptlets `<% // Kommentar %>` bzw. `<% /* Kommentar */ %>`.

Wie man in diesem Beispiel deutlich sehen kann, führt die Verwendung von Skripten (Scriptlets, Ausdrücken und Deklarationen) wieder zur verwirrenden Vermischung von HTML- und Java-Quellcode. Aus diesem Grund sollte man Scriptlets in JSP vermeiden.

Direktiven - directives

Die Aufgabe von Direktiven ist es Nachrichten an den JSP-Container zu senden. Es wird keinerlei Ausgabe erzeugt. Die gesendeten Parameter beeinflussen die Umwandlung der JSPs in Servlets durch den JSP-Container

Der allgemeine Aufbau eines Tags mit Direktive sieht folgendermaßen aus:
`<%@ directive attribut1="wert1" ... %>`

Es existieren drei Direktiven

- page
- include
- taglib

page-Direktive

Mittels der page-Direktive kann man das Verhalten einer JSP (einer einzelnen Seite) durch Angabe eines oder mehrerer Attribut-Wert-Paare festlegen. Mit Ausnahme des `import`-Attributes darf jedes Attribut nur einmal innerhalb einer JSP definiert werden.
Die meist gebrauchten Attribute sind wahrscheinlich `session` und `import`.

`language` definiert die Programmiersprache, die in den Scriptlets, Ausdrücken und Vereinbarungen und den mittels der include-Direktive (siehe unten) verwendet wird.
Derzeit ist nur "java" möglich, daher ist der Default-Wert: `<%@page language="java" %>`

`extends` definiert eine Java-Klasse die Superklasse dieser JSP sein soll. Sie muss den Vorgaben der JSP-Spezifikation genügen.
Default-Wert (Tomcat): `<%@page extends="HttpJspBase" %>`
Verändern dieses Attribut sollte sorgfältig durchdacht werden, da es sich um einen tiefen Eingriff in den Übersetztungsprozess des JSP-Containers darstellt!

`import` definiert die Typen/Packages die innerhalb der JSP verfügbar sein sollen. Die Syntax entspricht der eines „normalen" Java-Programmes.
`import` ist das einzige Attribut von `page` das mehrmals angegeben werden darf! Man kann daher die Werte entweder durch Kommata getrennt angeben oder die `page`-Direktive mehrmals mit dem Attribut import aufrufen.
Default-Wert:

```
<%@page import="java.lang.*, javax.servlet.*, javax.servlet.jsp.*" %>
<%@page import="javax.servlet.http.*" %>
```

`session` definiert ob die JSP an einer Session teilnehmen soll.
Default-Wert: `<%@page session="true" %>`
Dies entspricht dem Aufruf: `HttpSession session=request.getSession(true);`

`buffer` definiert die Größe des Puffers des `JspWriter` "`out`", in dem die JSP-Ausgaben zwischengespeichert werden, bevor sie an den Client abgeschickt werden.
Default-Wert: `<%@page buffer="8kb" %>`
Anmerkung: kein Pufferspeicher kann durch `buffer="none"` definiert werden.

`autoFlush` definiert ob die gepufferte Ausgabe automatisch „geflusht", oder ob eine Exception geworfen werden soll. Es ist nicht erlaubt `autoFlush="true"` zu setzen wenn `buffer="none"`.
Default-Wert: `<%@page autoFlush="true" %>`

`isThreadSafe` definiert ob der JSP-Container bei dieser JSP das *SingleThreadModel* nicht zu implementieren braucht. Beim *SingleThreadModel* werden nicht wie normalerweise mehrere Threads einer JSP gestartet, sondern es werden mehrere Instanzen einer JSP (ein Pool) geschaffen. Die Verwendung des SingleThreadModel verringert die Performanz einer Web-Anwendung und sollte daher vermieden werden.
Default-Wert: `<%@page isThreadSafe="true" %>`

`info` definiert eine Information über die JSP. Der Attributwert entspricht dem Überschreiben der Methode `getServletInfo()` mit `return("…")` in einem Servlet.
Beispiel: `<%@page info="Das ist ein JSP-Beispiel" %>`

`contentType` definiert den MIME-Typ der Antwort und den Zeichensatz in dem der Text der JSP verfasst ist.
Default-Wert: `<%@page contentType="text/html; charset=ISO-8859-1" %>`

`errorPage` dient zur Behandlung von Fehlern. Fehler können entweder schon bei der Übersetzung oder erst bei der Ausführung der JSP auftreten. In beiden Fällen meldet der Server einen HTTP-Statuscode 500 Internal Server Error. Tritt der Fehler bei der Übersetzung auf, liefert eine Fehlermeldung wo im Code der Fehler aufgetreten ist. Handelt

es sich um einen Laufzeitfehler, wird eine Exception geworfen. Wird diese nicht innerhalb der JSP (im Skriptcode) abgefangen, dann wird auf eine JSP zur Fehlerbehandlung verzweigt, sofern eine angegeben wurde. Falls keine Fehlerseite angegeben wurde, wird eine Fehlermeldung mit StackTrace angezeigt. `errorPage` definiert die JSP, die bei Auftreten einer Exception aufgerufen und an die das `Throwable`-Objekt übermittelt werden soll. Bei dieser JSP muss `isErrorPage="true"` gesetzt sein - siehe nächstes Attribut. Hierbei ist zu beachten, dass wenn in der Seite, in der der Fehler auftritt, autoFlush="true" gesetzt ist und die Ausgabe schon „geflusht“ wurde, es sein kann, dass die Fehlerbehandlung nicht funktioniert.
Beispiel: `<%@page errorPage="doexception.jsp" %>`

`isErrorPage` definiert ob es sich um eine JSP zur Behandlung von Exceptions handelt. Wird `isErrorPage="true"` definiert, dann wird vom JSP-Container die implizite Variable `exception` vom Typ `Throwable` (siehe oben) instanziert.
Default-Wert: `<%@page isErrorPage="false" %>`

Zur Illustration sollen zwei JSPs dienen. Eine Seite in der ein Laufzeitfehler (Division durch Null) auftritt und zur Fehlerbehandlung eine andere JSP weiterleitet, die eine Fehlermeldung anzeigt.

```
<%@page session="false" errorPage="fehlerseite.jsp"%>

<html>
  <head>
    <title>JSP ( page - Division by zero )</title>
  </head>
  <body>
    <h1>Hallo Welt</h1>
    <%= 1/0 %>
  </body>
</html>
```

Beispiel 2-24
`laufzeitfehler.jsp`

```
<%@page isErrorPage="true"%>

<html>
  <head>
    <title>JSP ( page - isErrorPage )</title>
  </head>
  <body>
    <h1> 500 - Internal Server Error </h1>
    Exception: <%= exception %> <br />
    Message: <%= exception.getMessage() %>
  </body>
</html>
```

Beispiel 2-25
`fehlerseite.jsp`

include-Direktive

Bei der Besprechung der Servlets wurde der RequestDispatcher-Mechanismus vorgestellt, der es ermöglich andere HTML-Seiten, Servlets und JSPs mittels Include in die Ausgabe einzubinden bzw per Forward auf andere Seiten weiterzuleiten.
JSPs bieten zwei Möglichkeiten um Inhalte aus anderen Dateien einzufügen: die *include-Direktive* und die *include-Aktion*. Ihr Verhalten und ihrer Programmierung differieren grundsätzlich voneinander, da sie für verschiedene Einsatzbereiche gedacht sind.

Zunächst wird die include-Direktive besprochen (zur include-Aktion siehe später).

Die include-Direktive dient zum Einfügen von anderen lokalen Dateien. Diese Webseiten werden durch das Attribut file festgelegt: `<%@ include file="Datei" %>`. Bei der Dateiangabe darf es sich nur um eine (kontext-) relative Pfadangabe handeln. Die Datei muss also lokal gespeichert sein und muss Teil der Web-Applikation sein.
Der Datei-URI darf zwar mit "../" beginnen, doch die referenzierte Datei muss sich innerhalb des Dateibaumes der Web-Applikation befinden. Es ist nicht möglich über eine Kontext-relative Pfadangabe ("/..") oder durch mehrfache Verwendung über eine pfad-relative Pfadangabe ("../../..") aus der Web-Applikation heraus auf andere Dateien am Server zuzugreifen.
Die Auswertung des Tags geschieht zur Übersetzungszeit. Der JSP-Container ersetzt die include-Direktive durch den Inhalt der angegebenen Datei. HTML-Code der inkludierten Seite wird in `out.println()`-Befehle transformiert, JSP-Code wird in die den Java-Code der aufrufenden Seite eingebaut und wie der JSP-Code der aufrufenden Seite behandelt. Der Gültigkeitsbereich von lokalen Variablen und Objekten wird dadurch auf die aufgerufenen Seiten ausdehnt. Aufrufende und aufgerufene JSP besitzen daher ein gemeinsames page-Objekt und ein gemeinsames request-Objekt.
Durch diese Funktionsweise ist es folglich auch nicht erlaubt bzw. möglich dem file-Attribut einen veränderlichen Inhalt (z.B. eine String-Variable) zuzuweisen.

Das nächste Beispiel soll zeigen wie eine weitere JSP mit Code integriert und eine HTML-Seite inkludiert werden. Danach ist der vom JSP-Container erzeugte Servlet-Code (gekürzt) abgebildet. Es zeigt sich, dass

- die Variable i eine Instanzvariable der gesamten (resultierenden) JSP wird,
- die Variable a (von include.jsp) auch in der aufgerufenen JSP (test.jsp) ohne vorangegangene Deklaration verwendet werden kann.
- "text/plain" zum Content-Type der gesamten (resultierenden) JSP wird.

```
<html>
  <head>
    <title>JSP ( directive.include )</title>
  </head>
  <body>
    <h1>Hallo Welt</h1>
    <% int a=1; %>
    <%@ include file="test.jsp" %>
    <hr />
    <%@ include file="footer.html" %>
  </body>
</html>
```

Beispiel 2-26
include.jsp

```
<i> &copy; 2000 by Harald Kornfeil </i>
```

Beispiel 2-27
footer.html

```
<%@page contentType="text/plain" %>

<%! int i = 0;   %>

Dies ist die inkludierte Seite<br />
Der Wert von a ist: <%= a %> <br />
Der Wert von i ist: <%= i++ %> <br />
<% String param = request.getParameter("test");
  if ((param==null) || (param.equals("")))
    out.println("no parameter 'test' or no value");
  else
    out.println("test="+param);
%>
```

Beispiel 2-28
test.jsp

```
public class _0002fjsp_ … _0002ejspinclude_jsp_6 extends HttpJspBase {

         int i = 0;

    static {
    }
    public _0002fjsp_ … _0002ejspinclude_jsp_6( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse  response)
                            throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/plain");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                  "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
```

```
            out = pageContext.getOut();

                out.write("<html>\r\n  <head>\r\n    <title>JSP (
directive.include )</title>\r\n  </head>\r\n  <body>\r\n    <h1>Hallo
Welt</h1>\r\n    ");

                 int a=1;

                out.write("\r\n    ");
                out.write("\r\n\r\n");

                out.write("\r\n\r\nDies ist die inkludierte Seite<br />\r\nDer
Wert von a ist: ");
                out.print( a );
                out.write(" <br />\r\nDer Wert von i ist: ");
                out.print( i++ );
                out.write(" <br />\r\n");
                 String param = request.getParameter("test");
                  if ((param==null) || (param.equals("")))
                    out.println("no parameter 'test' or no value");
                  else
                    out.println("test="+param);
                out.write("\r\n");

                out.write("\r\n    <hr />\r\n    ");

                out.write("<i> &copy; 2000 by Harald Kornfeil </i>");

                out.write("\r\n  </body>\r\n</html>");
            // end

        } catch (Exception ex) {
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

Beispiel 2-29
vom JSP-Contrainer generierter Java-Servlet-Code

Bei der Entwicklung ist zu beachten, dass bei Verwendung der Include-Direktive die JSP-Spezifikation nicht vorschreibt, dass eine JSP, die eine include-Direktive enthält, vom JSP-Container neu übersetzt werden muss, falls eine von ihr inkludierte Seite verändert wird.

taglib-Direktive

Die Taglib-Direktive einer JSP deklariert, dass die JSP eine Tag Library nützt. Tag Libraries ermöglichen die Verwendung von Tags, die nicht standardmäßig Teil der JSP-Spezifikation sind.
Einer der Vorteile von JSPs seit der Spezifikation 1.1 war, gegenüber anderen vergleichbaren Techniken, die Möglichkeit selbst definierte Tags zu implementieren und sie in Tag Libraries zur Verwendung in JSPs zur Verfügung zu stellen.
Die JSP-Spezifikation 1.1 definierte die Interfaces `Tag` und `BodyTag` zur Programmierung von einfachen (leeren) Tags und Tags mit Anfang- und End-Tag, in deren Rumpf (Body) HTML- und JSP-Code eingeschlossen werden kann. Die JSP-Spezifikation 1.2 führte zusätzlich das

Interface `IterationTag`, sowie das Interface `TryCatchFinally` ein. Letzteres kann einer Klasse hinzugefügt werden, die die Interfaces `Tag`, `IterationTag` oder `BodyTag` implementiert und dient zum Abfangen von Exceptions. Als Basis für selbst geschriebene Tag-Handler stellt die JSP-API die von den oben erwähnten Interfaces abgeleiteten Klassen TagSupport und BodyTagSupport zur Verfügung.
Da die Programmierung eigener Tags mit einigem Aufwand verbunden ist, wurden mit der JSP-Spezifikation 2.0 um das Interface SimpleTag und die Klasse SimpleTagSupport erweitert, die die Realisierung wesentlich vereinfachen sollen.

Wie später noch gezeigt wird stellen die JSP-Aktionen eine Art von Standard-Tag-Library für Zugriff auf Objekte dar. Um den Java-Code in JSPs zu reduzieren fehlte jedoch eine Tag-Unterstützung für einfache Objektmanipulation und Programmflusskontrolle. Weiters wurden von der Entwicklergemeinde Tags für Zugriff auf SQL-Datenbanken und für die Behandlung von XML-Daten gewünscht. Diese Funktionen wurden von einer Arbeitsgruppe im Rahmen des Java Community Process (Java Specification Request JSR052) als Java Standard Tag Library (JSTL) standardisiert. Die JSTL beherrscht eine so genannte *Expression Language* (EL) ein, die innerhalb der JSTL-Tags verwendet werden konnte. Durch einfache Ausdrücke werden ein einfacher Zugriff und eine Manipulation von Objekten möglich.
Aufgrund der Einfachheit und der Vorteile die die EL für die Trennung von Präsentation von Programmcode bot, wurde die EL als fixer Bestandteil in die JSP Spezifikation 2.0 aufgenommen und gilt bei JSP2.0 kompatiblen JSP-Containern für die gesamte JSP und nicht nur für bestimmte Tags.

Mit JSP-Spezifikation 2.0 erhielt die Taglib-Direktive noch eine weitere Aufgabe: die Deklaration von JSP-Funktionen. Funktionen ermöglichen einen einfachen Aufruf *statischer Methoden* von Java-Klassen. Gemeinsam mit der Verwendung der EL wird die Auslagerung von Code stark vereinfacht und die Verwendung von Scriptlets kann vermieden werden.

Programmierung selbst definierter Tags [32]

Die Funktionalität der Tags, befindet sich in Java-Klassen, wobei jedem Tag eine eigene Klasse (eine JavaBean) entspricht. Die Zuordnung des Tags und seiner Attribute zu den Java-Klassen wird mittels eines Tag-Library-Deskriptors definiert. Dies ist eine Text-Konfigurationsdatei, in der die Daten in einer XML-konformen Struktur abgespeichert sind.

Mittels der taglib-Direktive wird angegeben, welche Tag-Library in der JSP verwendet werden soll (genauer: es wird der zum Übersetzten notwendige Tag-Library-Deskriptor angegeben) und gleichzeitig ein Präfix zur Unterscheidung mehrerer Tag Libraries definiert, da diese gleichlautende Tags enthalten könnten:
`<%@ taglib uri="`*tagLibraryURI*`" prefix="`*tagPrefix*`" %>`

Das URI-Attribut enthält entweder einen absoluten oder einen relativen URI, zur eindeutigen Identifikation des Tag-Library-Descriptors (TLD). Dieser URI kann entweder eine gültige Internet-Adresse sein, oder ein Identifier für einen in web.xml definierten Tag-Library-Deskriptor.
In einer JSP können Tags verschiedener Tag-Libraries verwendet werden. Um diese unterschieden zu können, bedient man sich des Konzepts der XML-Namespaces. Der im `prefix`-Attribut angegebene String muss allen Elementnamen der ihm zugehörigen Tag-Library (durch einen Doppelpunkt voneinander getrennt) vorangestellt werden.
Die Präfixes `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, and `sunw` sind als *reserved* definiert und dürfen nicht verwendet werden. Weiters sind leere Präfixes laut Spezifikation nicht erlaubt.

Als Beispiel soll ein einfacher Tag zur Ausgabe von Datum und/oder Zeit dienen. Der Tag-Name ist `serverzeit` und hat zwei Attribute: `datum` und `uhrzeit`. Werden diesen die Werte „`false`“ oder „`no`“ zugewiesen oder wird das Attribut gar nicht angeführt, dann wird das Datum bzw. die Urzeit nicht ausgegeben – in allen anderen Fällen schon. Hat der Parameter test demnach einen Wert ungleich false oder no wird die Zeit ausgegeben.

```
<%@ taglib uri="bsptag.tld" prefix="bt" %>

<html>
  <head>
    <title>JSP ( Taglib ) </title>
  </head>
  <body>
    <h1>Hallo Welt</h1>
    <hr />
    Datum: <bt:serverzeit datum="<%request.getParameter("datum");%>" /> <br />
    Zeit: <bt:serverzeit uhrzeit="<%request.getParameter("zeit");%>" /> <br />
    <hr />
  </body>
</html>
```

Beispiel 2-30
`taglib.jsp` *mit Tag* `<bt:serverzeit>`

In der Direktive kann ein URI auch in Form einer absoluten Adresse oder eines Platzhalters angegeben werden. Zu beachten ist hierbei, dass es sich tatsächlich in beiden Fällen um einen Platzhalter handelt, der in web.xml definiert werden muss.
Die Aufrufe der taglib-Direktive können z.B. entweder so
`<%@ taglib uri="http://Servername/Verzeichnis/bsptag.tld" prefix="bt" %>`
oder so aussehen `<%@ taglib uri="beispieltag" prefix="bt" %>`

In jedem Fall muss sich in `/WEB-INF/web.xml` ein Eintrag befinden, der angibt wo der TLD zu finden ist. Hier befindet er sich im Verzeichnis `/WEB-INF/` und ist dadurch von außen nicht lesbar.

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <taglib>
        <taglib-uri>http://Servername/Verzeichnis/bsptag.tld</taglib-uri>
        <taglib-location>/WEB-INF/bsptag.tld</taglib-location>
    </taglib>

    <taglib>
        <taglib-uri>beispieltag</taglib-uri>
        <taglib-location>/WEB-INF/bsptag.tld</taglib-location>
    </taglib>
</web-app>
```

Beispiel 2-31
Taglib-Einträge in `/WEB-INF/web.xml`

Der TLD des Beispiels sieht so aus:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>Datum</shortname>
  <tag>
    <name>serverzeit</name>
    <tagclass>at.ac.akhwien.mvc.model.BeispielTag</tagclass>
    <info>Zeit und Datum des Servers angeben</info>
    <attribute>
      <name>uhrzeit</name>
      <required>no</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>datum</name>
      <required>no</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

</taglib>
```

Beispiel 2-32
Taglib-Deskriptor `bsptag.tld`

In diesem TLD wird nur ein Elementname der Tag-Library (serverzeit) definiert und mit der JavaBean (`at.ac.akhwien.mvc.model.BeispielTag`), welche die Programmlogik für diesen Tag enthält, verknüpft. Weiters werden auch die Attribute (`uhrzeit`, `datum`) für den Tag vereinbart. Die Attribute müssen nicht zwingend vorhanden sein (`required` = `no`, Defaultwert = `no`) und der Wert des Attributes darf zur Ausführungszeit dynamisch übergeben werden (`rtexprvalue` = `true`, Defaultwert = `false`). Es gibt noch einige Tag-Library-Deskriptor-Elemente mehr um das Verhalten der Tags zu beeinflussen, wie z.B. ob Attributwerte zur Ausführungszeit veränderlich sein dürfen und viele andere mehr - siehe dazu die Tag-Library-Deskriptor-DTD

Implementierung einfacher Tags ohne Rumpf

Bei den Klassen, die die Tag-Funktionen ausführen, handelt es sich um Java Beans. Eine Java-Bean ist eine gekapselte Programm-Komponente. Die in ihr gespeicherten Daten (ihre *Eigenschaften* bzw. *Properties*) können von Außen nur über `set`-Methoden geschrieben und über `get`- oder `is`-Methode gelesen werden. Heißt eine Eigenschaft z.B. `xyz` dann können andere Java-Programme - in diesem Falle der JSP-Container - in die Bean mittels Introspection hineinsehen und vergleichen ob eine Schreib-Methode namens `setXyz` und/oder Abfrage-Methoden namens `getXyz` bzw. `isXyz` existieren. Die Methode `isXyz` ist nur zur Abfrage von Eigenschaften vom Typ `Boolean` (wahr/falsch) geeignet, während get- und set-Methoden für jeden Variablen- oder Objekttyp verwendet werden können. Dabei ist zu beachten, dass `setXyz` den gleichen Typ schreibt wie `getXyz` ausließt! Welcher Typ zur internen Speicherung der Eigenschaft verwendet wird ist davon aber unabhängig!

Die JSP-API stellt Klassen zur Verfügung die die Tag-Interfaces implementieren. Von diesen Klassen werden die eigenen Tag-Handler abgeleitet.

Vom Interface Tag bzw. IterationTag ist die Klasse TagSupport abgeleitet. Das Interface Tag vererbt die Methoden doStartTag und doEndTag. Das Interface IterationTag erbt die beiden und fügt doAfterBody hinzu.
Ursprünglich war die Klasse TagSupport für die Programmierung von "einfacheren" Tags gedacht. Diese Aufgabe hat mit JSP 2.0 die Klasse SimpleTagSupport übernommen bei der nur die Methode doTag implementiert werden muss.
Für komplexere Aufgaben, wo Daten im Rumpf zwischen Anfangs- und End-Tag verarbeitet werden müssen, bietet die vom Interface BodyTag abgeleitete Klasse BodyTagSupport die umfassendsten Möglichkeiten. BodyTag ist von IterationTag abgeleitet und führt zusätzlich die Methode doInitBody ein.

Um einen eigenen Tag zu schreiben benötigt man daher nur eine von TagSupport abgeleitete Klasse, welche die Methoden doStartTag() bzw. doEndTag()überschreibt. Die Methode doStartTag() enthält die Programmlogik, die ausgeführt werden soll, wenn der entsprechende Tag in der JSP geöffnet wird, doEndTag()gilt dementsprechend für Ende-Tags.
Die Attribute des Start-Tags werden der Bean mit Hilfe der set-Methoden übergeben, deren Property-Namen den im TLD angegebenen Attributnamen entsprechen (siehe auch unten).
Mit den Return-Werten von doStartTag()wird dem JSP-Container mitgeteilt ob der Rumpf ausgewertet werden soll (EVAL_BODY_INCLUDE) oder ob die Auswertung des Body übersprungen werden soll (SKIP_BODY) bzw. im Falle von doEndTag() ob der Rest der JSP abgearbeitet werden soll (EVAL_PAGE) oder nicht (SKIP_PAGE).

`doStartTag()`	`EVAL_BODY_INCLUDE`	Rumpf abarbeiten
	`SKIP_BODY`	Rumpf überspringen
`doEndTag()`	`EVAL_PAGE`	restliche JSP abarbeiten
	`SKIP_PAGE`	Abarbeitung der JSP abbrechen

Tabelle 2-2
Rückgabewerte von `doStartTag()` *und* `doEndTag()`

```
package at.ac.akhwien.mvc.model;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;


public class BeispielTag extends TagSupport
{
  private boolean zeitZeigen;
  private boolean datumZeigen;


  public BeispielTag()
  {
    // Konstruktoraufruf von BeispielTag
  }

  public void setUhrzeit (String zeitZeigen_neu)
  {
    zeitZeigen=true;
    if(zeitZeigen_neu.equals("false") || zeitZeigen_neu.equals("no") )
      zeitZeigen=false;
  }
```

```
  public void setDatum (String datumZeigen_neu)
  {
    datumZeigen=true;
    if(datumZeigen_neu.equals("false") || datumZeigen_neu.equals("no") )
      datumZeigen=false;
  }

  public int doStartTag() throws JspException
  {
    String datumsFormatString="";
    String datumsString="";
    java.text.SimpleDateFormat datumsFormat = null;
    java.util.Calendar jetzt = null;

    if (datumZeigen)
      datumsFormatString=datumsFormatString+"d. MMMMM yyyy";

    if ((datumZeigen) && (zeitZeigen))
      datumsFormatString=datumsFormatString+", ";

    if (zeitZeigen)
      datumsFormatString=datumsFormatString+"HH:mm";

    datumsFormat = new java.text.SimpleDateFormat(datumsFormatString);
    jetzt = new java.util.GregorianCalendar();

    datumsString=datumsFormat.format(jetzt.getTime());
    try
    {
      pageContext.getOut().write(datumsString);
    }
    catch (java.io.IOException e)
    {
      throw new JspTagException(e.getMessage());
    }

    return SKIP_BODY;
  }

  public int doEndTag() throws JspException
  {
    return EVAL_PAGE;
  }

}
```

Beispiel 2-33
Implementation des Tags mit TagSupport: `BeispielTag.java`

Mit der Klasse SimpleTagSupport gestaltet sich die Implementation einfacher:

```
package at.ac.akhwien.mvc.model;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class BeispielTag extends SimpleTagSupport
{
  private boolean zeitZeigen;
  private boolean datumZeigen;

  public BeispielTag()
  {
    // Konstruktoraufruf von BeispielTag
  }
```

```
  public void setUhrzeit (String zeitZeigen_neu)
  {
    zeitZeigen=true;
    if(zeitZeigen_neu.equals("false") || zeitZeigen_neu.equals("no") )
      zeitZeigen=false;
  }

  public void setDatum (String datumZeigen_neu)
  {
    datumZeigen=true;
    if(datumZeigen_neu.equals("false") || datumZeigen_neu.equals("no") )
      datumZeigen=false;
  }

  public void doTag() throws JspException, java.io.IOException
  {
    String datumsFormatString="";
    String datumsString="";
    java.text.SimpleDateFormat datumsFormat = null;
    java.util.Calendar jetzt = null;

    if (datumZeigen)
      datumsFormatString=datumsFormatString+"d. MMMMM yyyy";

    if ((datumZeigen) && (zeitZeigen))
      datumsFormatString=datumsFormatString+", ";

    if (zeitZeigen)
      datumsFormatString=datumsFormatString+"HH:mm";

    datumsFormat = new java.text.SimpleDateFormat(datumsFormatString);
    jetzt = new java.util.GregorianCalendar();

    datumsString=datumsFormat.format(jetzt.getTime());
    try
    {
      pageContext.getOut().write(datumsString);
    }
    catch (java.io.IOException e)
    {
      throw new JspTagException(e.getMessage());
    }
  }
}
```

Beispiel 2-34
Implementation des Tags mit SimpleTagSupport: `BeispielTag.java`

Groß und Kleinschreibung von Taglib-Bean, TLD und JSP müssen aufeinander abgestimmt sein. Attribute eines Custom-JSP-Tags sollten (da es sich um XML-Attribute handelt)- generell klein geschrieben werden.

betrachten wir z.B. folgende set-Methode einer Taglib-bean:

```
  public void setUhrzeit (String zeitZeigen_neu)
  {
    zeitZeigen=true;
    if(zeitZeigen_neu.equals("false") || zeitZeigen_neu.equals("no") )
      zeitZeigen=false;
  }
```

Beispiel 2-35
JSP `taglib.jsp` mit taglib-Direktive zum Aufruf der Beispiel Taglib

der erste Buchstabe nach set muss groß geschrieben werden!
SetUhrzeit ist natürlich nicht gleich setUhrZeit!

Heißt die Methode setUhrzeit, müssen TLD und JSP folgendermaßen aussehen:

```
...
    <attribute>
      <name>uhrzeit</name>
      <required>no</required>
    </attribute>
...
```

```
...
    Zeit: <bt:serverzeit uhrzeit="yes" /> <br />
...
```

Eine falsche Angabe im TLD

```
...
    <attribute>
      <name>Uhrzeit</name>
      <required>no</required>
    </attribute>
...
```

würde zu folgender Fehlermeldung führen

```
Error: 500
Internal Servlet Error:
Cannot find any information on property 'Uhrzeit' in a bean of type
'at.ac.akhwien.mvc.model.BeispielTag '
```

Und auch eine falsche Angabe in der JSP (z.B. UHRZEIT oder Uhrzeit) ist nicht gültig.

```
Error: 500
Internal Servlet Error: Attribute Uhrzeit invalid according to the
specified TLD
```

Implementierung komplexer Tags mit Rumpf

Soll ein Tag implementiert werden bei dem der Rumpf ausgewertet wird, dann ist das mit den Klassen TagSupport und BodyTagSupport möglich. Zusätzlich zu den Methoden doStartTag() bzw. doEndTag() existieren hier noch die Methoden doInitBody() und doAfterBody().

Die Rückgabewerte unterscheiden sich nicht nur von den vorher erwähnten. Sie haben sich auch mit JSP-API 1.2 geändert:

`doStartTag()`	`EVAL_BODY_BUFFERED`	Rumpf abarbeiten
	`SKIP_BODY`	Rumpf überspringen
`doInitBody()`	-	Kein Rückgabewert
`doAfterBody()`	`EVAL_BODY_AGAIN`	Rumpf noch einmal abarbeiten
	`SKIP_BODY`	Abarbeitung des Rumpfes abbrechen
`doEndTag()`	`EVAL_PAGE`	restliche JSP abarbeiten
	`SKIP_PAGE`	Abarbeitung der JSP abbrechen

Tabelle 2-3

Rückgabewerte von `doStartTag(), doInitBody(), doAfterBody und doEndTag()`

Zugriff auf Objekte der JSP und Skript Variablen

Um von Tag-Klassen aus auf die in den verschiedenen Scopes der Web-Applikation gespeicherten Objekte zugreifen zu können benötigt man eine Zugriffsmöglichkeit auf den PageContext. Die Klassen TagSupport und BodyTagSupport haben dazu das Feld pageContext. Bei Verwendung der Klasse SimpleTagSupport kann man über die Methode getJspContext() den PageContext erhalten.
Mit den Methoden `getAttribute (`*Name*`, `*Scope*`)` und `setAttribute(`*Name*`, `*Objekt*`, `*Scope*`)` können Objekte im angegebenen Scope gelesen und abgelegt werden.

Es gibt aber auch die Möglichkeit so genannte Skript-Variablen zu vereinbaren, die dann anderen Tags oder Scriptlets zur Verfügung stehen. Sie sind Variablen der Methode _jspService() und keine Objekte die in anderen impliziten JSP-Objekten gespeichert sind. Ihre Definition geschieht in TEI-Klassen (TagExtraInformation) durch Überschreiben der Methode `VariableInfo[] getVariableInfo(TagData data)`. Diese liefert dem Tag-Handler die Informationen

- Name der Variablen,
- Typ bzw. Klassenname der Variablen,
- ob die Variable neu eingeführt wird und
- ab wann bzw. wo sie gültig ist.

Der Gültigkeitsbereich der Skriptvariablen hat nichts mit dem schon erwähnten *Scopes* bei impliziten Objekten zu tun:

`VariableInfo.AT_BEGIN`: Gültig ab dem Start-Tag
`VariableInfo.AT_END`: Gültig ab dem End-Tag
`VariableInfo.AT_NESTED`: Gültig zwischen dem Start- und dem End-Tag

TEI-Klassen werden im TLD mit dem Tag `<teiclass>` deklariert.

Exceptions in Tag-Library-Klassen

Wie die Methoden `doStartTag()` bzw. `doEndTag()` im obigen Beispiel werfen auch Methoden `doInitBody()` bzw. `doAfterBody()` JSP-Exceptions, die an die JSP weitergereicht werden. Andere Fehler müssen vom Programm mittels `try-catch` abgefangen werden und können als JspTagException „weiter geworfen“ werden. Dabei hilft das Interface `TryCatchFinally` mit den Methoden `doCatch()` und `doFinally()`.

Funktionen

Die Java Standard Tag Library (JSTL) hat eine Expression-Language EL eingeführt. Mit Aufnahme der EL in die JSP-Spezifikation 2.0 wurden Funktionen als Bestandteil der EL definiert.
Die Syntax der Funktionsaufrufe entspricht der der EL-Ausdrücke. Sie sind in mit einem $-Zeichen beginnende geschwungene Klammern eingeschlossen. Wie bei Tag-Libraries ist noch ein Namespace-Präfix zur Unterscheidung mehrer Tag-Libraries vorangestellt:
`${ prefix:Funktion() }`. Funktionen können überall dort verwendet werden wo auch EL-Ausdrücke oder Scriptlets erlaubt sind.

In Anlehnung an das Tag-Library-Beispiel zeigt das nächste Beispiel die Implementierung einer Funktion zur Anzeige von Zeit und Datum.
Der Funktion kann ein Parameter übergeben werden. Der Wert des Parameters kann entweder `"datum"` oder `"zeit"` sein. Dementsprechend liefert die Funktion das Datum oder die Urzeit. In allen anderen Fällen liefert die Funktion den Wert null.

```
<%@ page session="false" %>
<%@ taglib uri="bspfunction.tld" prefix="bt" %>

<html>
  <head>
    <title>JSP ( Taglib ) </title>
  </head>
  <body>
    <h1>Hallo Welt</h1>
    <hr />
    Datum: ${bt:serverzeit("datum")} <br />
    Zeit: ${bt:serverzeit("zeit")} <br />
    <hr />
  </body>
</html>
```

Beispiel 2-36
JSP `bspfunction.jsp` *mit JSP-Funktion* `${bt:serverzeit}`

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">

  <taglib>
    <taglib-uri> bspfunction.tld </taglib-uri>
    <taglib-location> /WEB-INF/bspfunction.tld </taglib-location>
  </taglib>

</web-app>
```

Beispiel 2-37
Taglib-Einträge in `/WEB-INF/web.xml`

Der TLD des Beispiels enthält die Definition der Funktion:

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
    jsptaglibrary_2_0.xsd" version="2.0">

   <tlib-version>1.0</tlib-version>
   <short-name>bt</short-name>
   <uri>bspfunction</uri>


   <function>
     <name>serverzeit</name>

     <function-class>
```

```
       at.ac.akhwien.mvc.taglibs.bspfunction
     </function-class>

     <function-signature>
       java.lang.String serverzeit(java.lang.String)
     </function-signature>

   </function>

</taglib>
```

Beispiel 2-38
Die TagLibrary `bspfunction.tld`

```
package at.ac.akhwien.mvc.taglibs;


public class bspfunction
{

  public static String serverzeit(String was)
  {

    String datumsFormatString="";
    String datumsString="";
    java.text.SimpleDateFormat datumsFormat = null;
    java.util.Calendar jetzt = null;

    if (was.equals("datum"))
    {
      datumsFormatString=datumsFormatString+"d. MMMMM yyyy";
    }
    else if (was.equals("zeit"))
    {
      datumsFormatString=datumsFormatString+"HH:mm";
    }
    else
    {
      return (null);
    }
    datumsFormat = new java.text.SimpleDateFormat(datumsFormatString);
    jetzt = new java.util.GregorianCalendar();

    datumsString=datumsFormat.format(jetzt.getTime());

    return (datumsString);
  }

}
```

Beispiel 2-39
Die Implementation der JSP-Funktion `bspfunction.java`

Für die Programmierung einer Funktion existieren in der JSP-API weder eine Klasse noch ein Interface von dem die Klasse abgeleitet werden muss. Funktionen werden als statische öffentliche Methoden implementiert und im TLD deklariert.
Dadurch besitzen die Klassen aber keinerlei logische Verbindung mit dem JSP-Container und es ist daher nicht möglich in Funktionen auf Objekte oder Skript-Variablen der JSP-Applikation zuzugreifen. Alle notwendigen Informationen müssen als Parameter der Funktion übergeben werden.

Aktionen - actions

Aktionen könnte man als eine Art standardmäßige Tag-Library sehen, da sie im Gegensatz zu den Skript-Elementen und Direktiven, eigentlich keine neue Funktionalität bieten, sondern nur eine Erleichterung im Umgang mit Techniken, die sonst vom Programmierer implementiert werden müsste. Auch die Syntax von Aktionen ist den Tags von Tag-Libraries `<%jsp: ... %>` sehr ähnlich.

Aktionen bieten Tags für das leichtere Weiterleiten und Inkludieren anderer HTML-Seiten bzw. JSPs und zum einfachen Umgang mit JavaBeans und Web-Browser-Plugins.

Zum Weiterleiten auf und Einfügen von WWW-Ressourcen gibt es die forward-Aktion und die include-Aktion.

Forward-Aktion

Die Syntax der Standardaktionen ist XML-konform und bietet die Möglichkeit zur Übergabe von Parametern. Diese können im Rumpf des Tags angegeben werden:

```
<jsp:forward page="relativePfadangabe">
  <jsp:param name="Ausdruck" value="Ausdruck"  />
  [optional weitere<jsp: param>]
</jsp:forward>
```

Die Parameterangaben werden in einem QueryString übergeben.

Sollen keine Parameter übergeben werden, dann entspricht die Syntax einem leeren Element: `<jsp:forward page="relativePfadangabe" />`.

Die `forward`-Aktion entspricht der `forward()`–Methode des `RequestDispatcher`-Mechanismus (siehe oben). Daher erfolgt die Auswertung der `forward`-Aktion zum Anfragezeitpunkt. Das Attribut `page` muss daher keinen fixen Wert enthalten, sondern kann auch zur Ausführungszeit über ein Script (einen Ausdruck) angegeben werden.

Das folgende Beispiel zeigt wie mit Hilfe der `forward`-Aktion eine dem `RequestDispatcher`-Beispiel vergleichbare Funktionalität sehr einfach verwirklicht werden kann. Die Seite `input.html` ruft die JSP `forward.jsp` entweder mit oder ohne Parameter `test` auf. Wird `forward.jsp` ohne Parameter aufgerufen, erfolgt ein Forward zurück auf `input.html`. Wird ein Parameter an die JSP übergeben, wird auf die Seite `output.jsp` weitergeleitet, welche sämtliche im request-Objekt enthaltenen Parameter und Attribute auflistet.
`Die JSP forward.jsp` fungiert als *Controller* ohne Ausgabe und entscheidet, welche *View* zur Darstellung gelangen soll

```
<html>
  <head>
    <title>Input</title>
  </head>
  <body>
    Input<br />
    <ul>
      <li><a href="forward.jsp">forward.jsp</a></li>
```

```
      <li><a href="forward.jsp?test=qwerty">
          forward.jsp?test=qwerty</a></li>
    </ul>
  <body>
</html>
```

Beispiel 2-40
`input.html`

```
<%
    String param = request.getParameter("test");
    String seite = null;

    if ((param==null) || (param.equals("")))
    {
      seite="input.html";
    }
    else
    {
      seite="output.jsp";
    }
    request.setAttribute("nachname","Kornfeil");

%>

<jsp:forward page="<%= seite %>" >
  <jsp:param name="vorname" value="Harald" />
</jsp:forward>
```

Beispiel 2-41
Controller-JSP `forward.jsp`

```
<%@page import="java.util.*" %>

<% Enumeration aufzaehlung = null;
   String name;
   String wert;
%>

<html>
  <head>
    <title>Output</title>
  </head>
  <body>
    Parameter:
    <ol> <% aufzaehlung = request.getParameterNames();
       for(;aufzaehlung.hasMoreElements();)
       {
         name = (String) aufzaehlung.nextElement();
         wert = request.getParameter( name);
         %> <li> <%= name %> :  <%= wert %> </li> <%
       } %>
    </ol>
    Attribute:
    <ol> <% aufzaehlung = request.getAttributeNames();
       for(;aufzaehlung.hasMoreElements();)
       {
         name = (String) aufzaehlung.nextElement();
         %> <li> <%= name %> </li> <%
       } %>
    </ol>

    <i>
      &copy; 2000 by
```

```
      <%= request.getParameter("vorname") %>
      <%= request.getAttribute("nachname") %>
    </i>
    <hr />
    Zur&uuml;ck zum / Back to
    <a href="start.html">Start</a>
  <body>
</html>
```

Beispiel 2-42
JSP `output.jsp`

Include-Aktion

Die Syntax der `include`-Aktion entspricht der Syntax der `forward`-Aktion. Auch der einzufügenden Seite können Parameter übergeben werden:

```
<jsp:inlcude page="relativePfadangabe" flush="true">
  <jsp:param name="Ausdruck" value="Ausdruck"  />
  [optional weitere<jsp:param>]
</jsp:include>
```

Die Syntax als leeres Element ohne Parameterangabe ist ebenso gültig:
`<jsp:include page="relativePfadangabe" flush="true"/>.`

Zusätzlich gibt es hier das Attribut `flush`, das angibt dass die aufrufende Seite vor dem Einfügen geflusht werden soll. Sollte nach dem Flush des Pufferspeichers eine `Exception` auftreten, so kann diese nicht durch eine in der `page`-Direktive der aufrufenden Seite angegebene Fehlerseite abgefangen werden!
Das Attribut flush muss entsprechend JSP-Spezifikation 1.1 immer angegeben werden und immer den Wert "true" erhalten. Erst mit der JSP-Spezifikation 1.2 ist es möglich auf die Angabe von flush zu verzichten. Der Default-Wert ist dabei "`false`".

Die `include`-Aktion entspricht der `include()`-Methode des `RequestDispatcher`-Mechanismus (siehe oben) und unterscheidet sich grundsätzlich von der `include`-*Direktive*. Im Gegensatz zum `file`-Attribut der `include`-Direktive muss das Attribut `page` der `include`-Aktion keinen fixen Wert enthalten. Wie bei der `forward`-Aktion wird der Parameter nicht zur Übersetzungszeit sondern zur Ausführungszeit ausgewertet und nur der Ausgabestrom der aufgerufenen Seite in die eigene Ausgabe eingefügt und nicht der Quelltext der aufgerufenen Seite in den eigenen JSP-Code übernommen. Für einen Datenaustausch müssen Daten im `request`-Objekt als Attribut übergeben werden. Aufrufende und aufgerufene JSP haben jeweils ein eigenes `page`-Objekt. Das `request`-Objekt wird an alle eingefügten Seiten weitergegeben!
Zusätzlich werden im request-Objekt vom Server Informationen über die aufrufende JSP als Attribute abgespeichert. Dazu wird im Folgenden `outputfooter.jsp` so erweitert, dass die übergebenen Parameter und Attribute dargestellt werden.

Ähnlich dem RequestDispatcher-Beispiel werden in diesem Beispiel, fall Parameter übergeben werden, zwei HTML-Seiten bei der Ausgabe inkludiert (`outputheader.html` und `outputfooter.jsp`) und zwischen ihren Aufrufen die Parameter und Attribute des request-Objektes ausgegeben. Werden keine Parameter mitgesendet, wird auf die Eingabeseite wieder zurückgeleitet (Eingabeseite nicht abgedruckt).

```
<html>
  <head>
    <title>Output</title>
  </head>
  <body>
```

Beispiel 2-43
outputheader.html

```
<%@ page import="java.util.*" %>

<% Enumeration aufzaehlung = null;
   String name;
   String wert;
%>

    <i>
      &copy; 2000 by
      <%= request.getParameter("vorname") %>
      <%= request.getAttribute("nachname") %>
    </i>
    <hr />
    Zur&uuml;ck zum / Back to <a href="start.html">Start</a>

    <hr />
    <ul>
      jsp:include - Parameter:
      <ol> <% aufzaehlung = request.getParameterNames();
        for(;aufzaehlung.hasMoreElements();)
        {
          name = (String) aufzaehlung.nextElement();
          wert = request.getParameter( name);
          %> <li> <%= name %> :  <%= wert %> </li> <%
        } %>
      </ol>

      jsp:include - Attribute:
      <ol> <% aufzaehlung = request.getAttributeNames();
        for(;aufzaehlung.hasMoreElements();)
        {
          name = (String) aufzaehlung.nextElement();
          %> <li> <%= name %> </li> <%
        } %>
      </ol>
    </ul>
  <body>
</html>
```

Beispiel 2-44
outputfooter.jsp

```
<%@ page import="java.util.*" %>

<% Enumeration aufzaehlung = null;
   String name;
   String wert;

   String param = request.getParameter("test");
   String seite = null;

   if ((param==null) || (param.equals("")))
   {
```

```
     %> <jsp:forward page="inputa.html" /> <%
   }
   else
   { %>
     <jsp:include page="outputheader.html" flush="true"/>

     Parameter:
     <ol> <% aufzaehlung = request.getParameterNames();
        for(;aufzaehlung.hasMoreElements();)
        {
          name = (String) aufzaehlung.nextElement();
          wert = request.getParameter( name);
          %> <li> <%= name %> :  <%= wert %> </li> <%
        } %>
     </ol>
     Attribute:
     <ol> <% aufzaehlung = request.getAttributeNames();
        for(;aufzaehlung.hasMoreElements();)
        {
          name = (String) aufzaehlung.nextElement();
          %> <li> <%= name %> </li> <%
        } %>
     </ol>

     <% request.setAttribute("nachname","Kornfeil"); %>

     <jsp:include page="outputfooter.jsp" flush="true" >
       <jsp:param name="vorname" value="Harald" />
     </jsp:include>
<% }
%>
```

Beispiel 2-45
include.jsp

Im davor angeführten forward-Beispiel fungierte `forward.jsp` als *Controller*-JSP dessen Aufgabe es war zu entscheiden an welche *View*-JSP die Ausgabe weitergeleitet werden soll. Bei der include-Version sind Programm-Logik und Ausgabe jedoch wieder stark vermischt.

Um die Vermischung von HTML und JSP/Java-Code zu vermindern bietet sich an die Funktionalität in einen Controller- und einen View-Teil zu aufzuspalten:

```
<%
    String param = request.getParameter("test");
    String seite = null;

    if ((param==null) || (param.equals("")))
    {
      seite="inputb.html";
    }
    else
    {
      seite="view.jsp";
    }
%>

<jsp:forward page="<%= seite %>" />
```

Beispiel 2-46
controller.jsp

```
<%@page import="java.util.*" %>

<%@page import="java.util.*" %>

<% Enumeration aufzaehlung = null;
   String name;
   String wert;
%>

<jsp:include page="outputheader.html" flush="true"/>

Parameter:
<ol> <% aufzaehlung = request.getParameterNames();
  for(;aufzaehlung.hasMoreElements();)
  {
    name = (String) aufzaehlung.nextElement();
    wert = request.getParameter(name);
    %> <li> <%= name %> :  <%= wert %> </li> <%
  } %>
</ol>
Attribute:
<ol> <% aufzaehlung = request.getAttributeNames();
  for(;aufzaehlung.hasMoreElements();)
  {
    name = (String) aufzaehlung.nextElement();
    %> <li> <%= name %> </li> <%
  } %>
</ol>

<% request.setAttribute("nachname","Kornfeil"); %>

<jsp:include page="outputfooter.jsp" flush="true" >
  <jsp:param name="vorname" value="Harald" />
</jsp:include>
```

Beispiel 2-47
`view.jsp`

Die nun erreichte Trennung von HTML und Programmcode ist zwar ein Schritt in die richtige Richtung, doch für ein Projektteam, das aus Designern und Programmierern besteht, die sich nur um ihren Teil kümmern sollen, ist der Wartungsaufwand durch die Durchmischung noch immer zu hoch. Um den Programmcode-Anteil weiter zu vermindern und die Wartbarkeit zu verbessern bietet sich an Tag-Libraries einzusetzen. Hier benötigte Funktionen wie Schleifen und Objekt-Zugriff sind in der JSTL enthalten, damit JSP-Programmierer nicht immer wieder das Rad neu erfinden müssen. Bevor näher auf die Möglichkeiten der JSTL eingegangen werden kann ist es notwendig zu wissen wie JavaBeans in JSPs eingesetzt werden, da die JSTL regen Gebrauch von JavaBeans machen.

JavaBeans-Aktionen

In der JSP-Spezifikationen wurden von Anfang an neben den JSP-Aktionen zum Weiterleiten und Einfügen von JSPs auch JSP-Aktionen definiert, die die Verwendung von JavaBeans vereinfachen sollen. Wie schon bei der Besprechung der Tag-Libraries erwähnt, handelt es sich bei JavaBeans um Komponenten zur Speicherung von Eigenschaften bzw. Properties, auf welche über Methoden zugegriffen werden kann, deren Namen durch den Property-Namen vorgegeben wird. Da es sich bei JavaBeans um normale Objekte handelt, können sie innerhalb des `page`-, `request`-, `httpsession`- oder `application`-Objektes abgelegt werden.

Um Bean-Aktionen benutzen zu können, bedarf es einer JavaBean. Die Bean; die in den folgenden Beispielen Verwendung findet, hat zwei Eigenschaften („`zahl`“ und „`test`“). Auf dieses kann von Außen nicht direkt zugegriffen werden (`private`) sondern nur über die set- und `get`-Methoden um die Eigenschaften zu schreiben und zu lesen (`public`):

```
package at.ac.akhwien.mvc.model;

public class BeispielBean
{
  // Eigenschaften
  private int zahl = 0;
  private String test = "";

  // Methoden
  public void setZahl(int zahl_neu)
  {
    this.zahl = zahl_neu;
  }

  public int getZahl()
  {
    return zahl;
  }

  public void setTest (String test_neu)
  {
    this.test = test_neu;
  }

  public String getTest ()
  {
    return this.test;
  }
}
```

Beispiel 2-48
`BeispielBean.java`

Um eine Bean innerhalb einer JSP zu verwenden muss sie angemeldet werden. Dies geschieht mittels

```
<jsp:useBean id="Name" class="Klassenname" scope="Gültigkeitsbereich" />
```

Das Attribut `id` gibt an unter welchem Namen die über das Attribut `class` angegebene Bean-Klasse innerhalb der JSP angesprochen werden soll. Über das Attribut `scope` kann ein Gültigkeitsbereich in dem die Bean verwendet werden kann, angegeben werden. D.h. es wird dadurch definiert in welchem impliziten JSP-Objekt die Bean abgespeichert werden soll.

Unter Berücksichtigung der oben genanten Besonderheiten, die beim Einfügen und Weiterleiten berücksichtigt werden müssen, ergeben sich für die scope-Werte folgende Gültigkeitsbereiche:

Scope	Gültig innerhalb ...
`page`	• der JSP • aller über die `include`-Direktive eingebundenen JSPs
`request`	• der JSP • aller über die `include`-Direktive eingebundenen JSPs • aller über die `include`-Aktion eingebundenen JSPs • aller JSPs auf die über die `forward`-Aktion weitergeleitet wurde
`session`	• aller JSPs und Servlets die an der derzeit für die JSP gültigen Session teilnehmen
`application`	• der gesamten Web-Applikation (wird im `ServletContext`-Object gespeichert)

Tabelle 2-4
Werte für scope und deren Gültigkeitsbereiche

Die Anmeldung der Bean erfolgt mit der JSP-Aktion `<jsp:useBean>` entweder als leeres Element:

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
             scope="page" />
```

als Element mit Rumpf

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
             scope="page" >
</jsp:useBean>
```

Innerhalb des Rumpfes können HTML-Code oder beliebige JSP-Elemente stehen, die ausgeführt werden sollen, wenn die Bean zum *ersten Mal* im angegebenen Scope instanziert wird – das heißt bei:

- `scope="page"` jedes Mal,
- `scope="request"` nur in der JSP in der die Bean als erstes im request-Objekt abgelegt wird,
- `scope="session"` nur in der JSP in der die Bean das erste mal im Session-Objekt abgelegt wird,
- `scope="application"` nur in der JSP einer Web-Applikation, in der die Bean als erste im ServletContext-Objekt abgelegt wird.

Bei der Verwendung der `include`-Direktive muss darauf geachtet werden, dass eine Bean nicht mehrmals unter derselben `id` im selben `scope` angemeldet wird, da dies nicht zulässig ist.

Zum Zugriff auf Eigenschaften stehen zwei Aktionen zur Verfügung:

- zum Speichern:
  ```
  <jsp:setProperty name="Name" property="Eigenschaftsname" value="Wert"
  param="Parametername"/>
  ```
- zum Lesen:
  ```
  <jsp:getProperty name="Name" property="Eigenschaftsname" />
  ```

Das Attribut `name` gibt den Namen der Bean an und entspricht dem in `<jsp:useBean />` mit dem Attribut `id` angegebenen Namen. Der Name der Eigenschaft auf die zugegriffen werden soll wird mit `property`, der zu schreibende Wert mit `value` angegeben.
Zum Attribut `param` siehe die ausführliche Besprechung im Abschnitt „Parameter und Beans“ unten.

Um den Wert 12345 in der Eigenschaft `zahl` der `BeispielBean` zu speichern kann man sich der JSP-Aktion `<jsp:setProperty name="bsp" property="zahl" value="12345" />` bedienen. Es kann aber auch aus Scriptlets über die set-Methode auf die Eigenschaft zugegriffen werden. Das Scriptlet-Äquivalent ist: `<% bsp.setZahl(12345); %>`

Zum Lesen des Wertes der Eigenschaft `zahl` der `BeispielBean` dient die JSP-Aktion `<jsp:getProperty name="bsp" property="zahl" />` anstelle des JSP-Ausdruckes `<%= bsp.getZahl() +"\n" %>`.

Um den Code lesbarer zu gestalten sollte den JSP-Aktionen der Vorzug gegeben werden.

page-scope

Im folgenden Beispiel wird eine Bean im `scope="page"` angelegt auf welche eine per include-Direktive eingefügte JSP zugreift:

```
<%@ page session="false" %>

<jsp:useBean id="bsp" class="AT.ac.akhwien.mvc.model.BeispielBean"
                      scope="page" >
  BeispielBean-Anmeldung
</jsp:useBean>

<html>
  <head>
    <title>useBean - scope="page"</title>
  </head>
  <body>
  <p>
    <jsp:setProperty name="bsp" property="zahl" value="12345" />
    zahl: <jsp:getProperty name="bsp" property="zahl" /> <br />

    <% bsp.setTest("qwerty"); %>
    test: <%= bsp.getTest() +"\n" %>
  </p>

  <%@ include file="pagebean_inc.jsp" %>

  </body>
</html>
```

Beispiel 2-49
`pagebean.jsp`

```
  <hr />
  <p>
    include: <br />
    zahl - vor set: <%= bsp.getZahl() +"\n" %> <br />
    <% bsp.setZahl(67890); %>
    zahl - nach set: <%= bsp.getZahl() +"\n" %> <br />

    test - vor set: <jsp:getProperty name="bsp" property="test" /> <br />
```

```
    <jsp:setProperty name="bsp" property="test" value="abcde" />
    test - nach set: <jsp:getProperty name="bsp" property="test" />
  </p>
```

Beispiel 2-50
pagebean_inc.jsp

Das Einfügen von `pagebean.jsp` per include-*Aktion* `<jsp:include page="pagebean.jsp" />` würde mit einer Fehlermeldung quittiert werden („Included servlet error: 500"), da `pagebean.jsp` ein eigenes page-Objekt ohne BeispielBean besitzt.

request-scope

Gibt man hingegen `scope="request"` beim Anlegen der Bean an, dann können Seiten, die das `Request`-Objekt über die Aktionen `include` oder `forward` erhalten haben, auf die Bean zugreifen. In diesem Fall jedoch müssen alle JSPs eine gleichnamige Bean im `request`-scope anmelden. Beim 2. und 3. Aufruf von `useBean` wird „BeispielBean-Anmeldung" nicht mehr ausgegeben.

```
<%@ page session="false" %>

<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="request" >
  BeispielBean-Anmeldung: requestbean.jsp
</jsp:useBean>

<jsp:setProperty name="bsp" property="zahl" value="12345" />
<% bsp.setTest("qwerty"); %>

<jsp:forward page="requestbean_fw.jsp" />
```

Beispiel 2-51
requestbean.jsp

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="request" >
  BeispielBean-Anmeldung: requestbean_fw.jsp
</jsp:useBean>

<html>
  <head>
    <title>useBean - scope="request"</title>
  </head>
  <body>
  <p>
    forward
    zahl vor set: <jsp:getProperty name="bsp" property="zahl" /> <br />
    <jsp:setProperty name="bsp" property="zahl" value="67890" />
    zahl nach set: <jsp:getProperty name="bsp" property="zahl" /> <br />

    test vor set: <%= bsp.getTest() +"\n" %> <br />
    <% bsp.setTest("abcde"); %>
    test nach set: <%= bsp.getTest() +"\n" %> <br />
  </p>

  <jsp:include page="requestbean_inc.jsp" flush="true" />

  </body>
</html>
```

Beispiel 2-52
requestbean_fw.jsp

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="request" >
BeispielBean-Anmeldung: requestbean_inc.jsp
</jsp:useBean>

  <hr />
  <p>
    include: <br />
    zahl - vor set: <%= bsp.getZahl() +"\n" %> <br />
    <% bsp.setZahl(12357); %>
    zahl - nach set: <%= bsp.getZahl() +"\n" %> <br />

    test - vor set: <jsp:getProperty name="bsp" property="test" /> <br />
    <jsp:setProperty name="bsp" property="test" value="asdfg" />
    test - nach set: <jsp:getProperty name="bsp" property="test" />
  </p>
```

Beispiel 2-53
`requestbean_inc.jsp`

session-scope

Wie schon bei den Servlets im Abschnitt Sitzungsmanagement besprochen, bietet die Servlet-API das HttpSession-Objekt um über mehrere Requests verteilte zusammengehörende Anfragen eines Clients serverseitig einfach verwalten zu können.

Die Teilnahme einer JSP an einer Sitzung wird über die page-Direktive mit dem Attribut session festgelegt.

In Verbindung mit im Rumpf der useBean-Aktion enthaltener Anweisungen bietet sich nun eine elegante Möglichkeit zur Überprüfung ob ein Client überhaupt schon an einer Sitzung teilnimmt (ob er sich z.B. schon angemeldet hat). Ist im aktuellen Session-Objekt noch keine BeispielBean vorhanden (welche nur nach erfolgreichem Login angelegt wird), dann wird die aktuelle Session für ungültig erklärt und der Client auf die Homepage der Web-Applikation weitergeleitet.

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" >
  <% session.invalidate(); %>
  <jsp:forward page="index.html" />
</jsp:useBean>

...
```

Ansonsten entspricht die Verwendung des Session Objekts dem schon oben erwähnten.

application-scope

Auf im application-Objekt abgelegte JavaBeans haben alle JSPs und Servlets einer Web-Applikation Zugriff. Hier können z.B. Datenbank-Verbindungen oder andere Ressourcen, welche für die gesamte Applikation wichtig sind, abgelegt werden.

Parameter und Beans

Die JSP-Aktion `<setProperty ... />` bietet nicht nur die oben gezeigte Möglichkeit vorgegebene Werte in einer JavaBean-Eigenschaft zu setzen. Sie hat auch die Fähigkeit dies automatisch mit Werten zu tun, die als Parameter an die JSP per `GET`- oder `POST`-Request übergeben wurden. Damit diese automatische Bevölkerung von Bean-Properties funktioniert, müssen Parametername und der Eigenschaftsname übereinstimmen.
Soll z.B. die Eigenschaft zahl der Beispielbean automatisch mit dem Wert eines Parameters ge- bzw. überschrieben werden, muss das `value`-Attribut weggelassen werden:

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="page" />

<jsp:setProperty name="bsp" property="zahl" />

<html>
  <head>
    <title>useBean param </title>
  </head>
  <body>
    <p>
      zahl: <jsp:getProperty name="bsp" property="zahl" /> <br />
    </p>
  </body>
</html>
```

Beispiel 2-54
`pagebean_param.jsp`

Wird nun die JSP mit dem Parameter/Wert-Paar `zahl=12345` aufgerufen, so wird diese in der Bean `bsp` im Property `zahl` mittels `<jsp:setProperty>` gespeichert und dann mittels `<jsp:getProperty>` ausgegeben.
Es muss beachtet werden, dass wenn kein Parameter und/oder kein Wert übergeben wird, oder ein Parameter anderen Namens übergeben wird kein neuer Wert in die Bean geschrieben wird. Der Wert der in der Bean gespeicherten Eigenschaft bleibt in solchen Fällen daher unverändert.

Über das Attribut `param` kann man einen Parameternamen angeben dessen Parameterwert in die durch property angegebene Eigenschaft geschrieben werden soll. Wird die `<jsp:setProperty>` Aktion des obigen Beispiels folgendermaßen erweitert,

```
...
  <jsp:setProperty name="bsp" param="wert" property="zahl" />
...
```

Beispiel 2-55
Auszug aus Beispiel `pagebean_param_wert.jsp`

dann kann die JSP nun mit dem Parameter/Wert-Paar `wert=12345` aufgerufen werden um die Ausgabe des übergebenen Parameterwertes zu erhalten.

Es gibt aber auch die Möglichkeit eine „Wildcard“ zu verwenden, um alle Eigenschaften mit Parametern gleichen Namens automatisch schreiben zu lassen:

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="page" />

<jsp:setProperty name="bsp" property="*" />

<html>
  <head>
    <title>useBean param </title>
  </head>
  <body>
    <p>
      zahl: <jsp:getProperty name="bsp" property="zahl" /> <br />
      test: <jsp:getProperty name="bsp" property="test" /> <br />
    </p>
  </body>
</html>
```

Beispiel 2-56
`pagebean_param_wild.jsp`

MVC mit JSP

Mit dem nun vorhandenen Wissen ist es sehr einfach, das schon von den Request-Dispatchern bekannte View-Controller-Konzept zu einem kompletten Model-View-Controller-Konzept auszubauen. Als Model dienen hierzu die JavaBeans.
Zur Demonstration soll das Controller-View-Beispiel mit der `BeispielBean` als Model erweiterter werden.

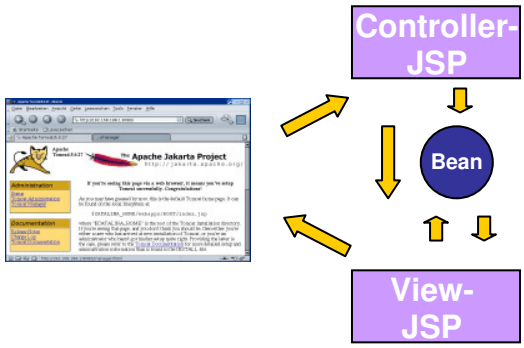

Abbildung 2-7
MVC mittels JSPs

Der Aufruf des Controllers geschieht in diesem Beispiel aber nicht mehr über einen statischen Link sondern über ein Formular:

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" />

<jsp:setProperty name="bsp" property="*" />

<html>
```

```
  <head>
    <title>Input - Session</title>
  </head>
  <body>
    Input - Session<br />

    <form action="controller.jsp" method="get" >
      Zahl: <input type="text" name="zahl" value="<%= bsp.getZahl() %>" />
      <br />
      Test: <input type="text" name="test" value="<%= bsp.getTest() %>" />
      <br />
      <input type="submit" />
    </form>

  <body>
</html>
```

Beispiel 2-57
input_view.jsp

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" />

<jsp:setProperty name="bsp" property="*" />

<%
    String param = bsp.getTest();
    String seite = null;

    if ((param==null) || (param.equals("")))
    {
      seite="input_view.jsp";
    }
    else
    {
      seite="output_view.jsp";
    }
%>
<jsp:forward page="<%= seite %>" />
```

Beispiel 2-58
controller.jsp

```
<jsp:useBean id="bsp" class="at.ac.akhwien. mvc.model.BeispielBean"
                      scope="session" />

<html>
  <head>
    <title>Output</title>
  </head>
  <body>
    <p>
      zahl: <jsp:getProperty name="bsp" property="zahl" /> <br />
      test: <jsp:getProperty name="bsp" property="test" /> <br />
    </p>
    <hr />
    Zur&uuml;ck zum <a href="input_view.jsp">Start</a>
  <body>
</html>

<% session.invalidate(); %>
```

Beispiel 2-59
output_view.jsp

Für dieses Beispiel wurde ein Session-Objekt gewählt um die Bean zu speichern. Damit dies funktioniert muss der Client an der Session teilnehmen. Falls der Client Cookies ausgeschaltet hat, würde dies zu unerwünschtem Verhalten führen.
Manche Servlet-Server führen automatisch ein „Fallback“ auf URL-Rewriting aus, wenn der Client Cookies ablehnt. Um sicherzustellen, dass das Beispiel in jedem Fall korrekt ausgeführt wird, könnte man hier die Methode `encodeURL()` verwenden, die nur dann ein URL-Rewriting vornimmt, falls der Client keine Cookies akzeptiert:

```
...
  <form action="<%= response.encodeURL("controller.jsp") %>" method="get" >
...
```

Bei der Verwendung der Methode `encodeURL()` ist auf richtige Groß- und Kleinschreibung zu achten, da in `javax.servlet.http.HttpServletResponse` auch die Methode `encodeUrl()` existiert, die mit der Servlet-Spezifikation 2.2 als „deprecated“ klassifiziert wurde, und daher nicht mehr verwendet werden sollte!

Das Beispiel würde auch funktionieren wenn die Bean im `request`-Objekt gespeichert würde. Allerdings lässt sich nur bei Verwendung des `session`-Objektes folgende Problematik zeigen, dass eine einmal gesetzte Eigenschaft nicht dadurch gelöscht werden kann, dass der für die Eigenschaft zuständige Parameter ohne Parameterwert an die JSP übergeben wird, da die set-Methode bei nicht vorhandenem Parameterwert vom JSP-Container nicht aufgerufen wird. Wird beispielsweise für `zahl` der Wert "1234" und für `test` nichts eingegeben, wird dadurch wieder die Seite `input_view.jsp` weitergeleitet - wie gewünscht mit dem eingegebenen Wert. Wenn die Zahl im Eingabefeld gelöscht wird und ein leeres Formular abschickt wird, erscheint erneut die Seite `input_view.jsp` - nun allerdings wieder mit dem anfangs eingegebenen Zahlenwert.
Wenn hier das `request`-Objekt statt dem session-Objekt verwendet würde, dann würde das `request`-Objekt nach der Weiterleitung ungültig und von der Servlet-Engine gelöscht werden. Für die nächste Anfrage würde der Server ein neues Request-Objekt zur Verfügung stellen.

Für solche Fälle sollte die Bean daher eine `Reset`-Methode zum Löschen aller Properties oder Methoden zum selektiven Löschen der Properties zur Verfügung stellen, die dann von der JSP aufgerufen werden kann.

Weiters lässt sich zeigen, dass beim automatischen Schreiben der Eigenschaften vom JSP-Container keine Typ-Überprüfung vorgenommen wird. Wird im obigen Fall z.B. im Formular das Feld für `zahl` mit eine Zeichenkette ausgefüllt, dann wird dies vom Server mit einer Fehlermeldung beantwortet. Es empfiehlt sich daher, dass die Methoden zum Lesen und Schreiben von Formular-Bean-Properties immer vom Typ `String` sein sollten und eventuell notwendige Umwandlungen in Bean-interne Datentypen innerhalb der `set-` und `get-` Zugriffsmethoden der Bean vorgenommen werden sollten!

2.5. Java Standard Tag Library - JSTL

Die Java Standard Tag Library ist eine Sammlung von JSP 1.2 Tags die grundlegende Funktionen zur Verfügung stellt, die oft in Web-Projekten benötigt werden und ermöglicht

JSP Autoren damit sich auf die Programmierung von applikationsspezifischen Code zu konzentrieren und nicht das Rad neu erfinden zu müssen.
Weiters zielt die JSTL darauf ab den Anteil an eingebetteten Programmcode in Form von Scriptlets weiter zu reduzieren und die Wartbarkeit der JSP zu verbessern.
Die JSTL bietet dazu Unterstützung zur Realisierung von

- Schleifen (Core-Bibliothek)
- Bedingungsabhängige Programmablaufsteuerung (Core-Bibliothek)
- Einfügen von Text in JSPs (Core-Bibliothek)
- Internationalisierung (I18n) (Format-Bibliothek)
- XML-Manipulation (XML-Bibliothek)
- Datenbank-Zugiff (SQL-Bibliothek)

Weiters bietet die JSTL eine so genannte Expression Language (EL) für seine Tags um Zugriff auf Objekte und deren Eigenschaften zu erleichtern. Diese kann bei JSP-Containern, die noch nicht der JSP-Spezifikation 2.0 entsprechen, nur innerhalb der JSTL-Tags verwendet werden. JSP 2.0 kompatible Container können die EL in der gesamten JSP nutzen, da die EL Teil der JSP-Spezifikation 2.0 ist.

Expression Language – EL

Die Expression Language ist an EcmaScript und XPath angelehnt. Dadurch sollte ihre Syntax sowohl für Java-Entwickler und HTML-Designer vertraut sein. Die EL dient zum Zugriff auf Objekte und ihre Eigenschaften und ermöglicht verschiedene logische und mathematische Operatoren auf sie anzuwenden.
EL-Ausdrücke werden in, mit einem $-Zeichen beginnende, geschwungene Klammern eingeschlossen: `${` *Ausdruck* `}`

Der Zugriff auf Objekte und ihre Methoden kann über zwei Operatoren geschehen die idente Ergebnisse liefern:

- Beim Punkt-Operator (`ausdruck-a.name-b`) werden Objekte und Attribute bzw. Methoden durch Punke getrennt aufgerufen.
- Eckige Klammern (`ausdruck-a[name-b]`) umschließen einen Methoden- oder Attributnamen. Der Zugriff über eckige Klammern ist eigentlich für `Collections` vom Typ `Map` gedacht, kann aber auch für Methoden verwendet werden.

So kann auf die Methode `getZahl()` des Objektes `bsp` durch den EL-Ausdruck `bsp.zahl` oder `bsp["zahl"]` zugegriffen werden.

Will man, wie im Rahmen der letzten Beispiel-Applikation gezeigt, den Wert einer Bean in einem Tag verwenden, benötigt man ein Scriptlet:

```
...
    Zahl: <input type="text" name="zahl" value="<%= bsp.getZahl() %>" />
...
```

Bei Verwendung der EL ist kein Scriptlet mehr nötig:

```
...
    Zahl: <input type="text" name="zahl" value="${bsp.zahl}" />
...
```

Für Berechnungen, Vergleiche oder logische Operationen können folgende Operatoren in EL-Ausdrücken verwendet werden:

Kategorie	Operatoren
Arithmetisch	`+`, `-`, `*`, `/` (oder `div`), % (oder `mod`)
Relational	`==` (oder `eq`), `!=` (oder `ne`), `<` (oder `lt`), `>` (oder `gt`), `<=` (oder `le`), `>=` (oder `ge`)
Logisch	`&&` (oder `and`), `||` (oder `or`), `!` (oder `not`)
Validierung	`Empty`

Tabelle 2-5
EL-Operatoren

Neben den über <jsp:useBean> deklarierten Objekten stehen den JSP-Entwicklern elf weitere implizite EL-Objekte zur Verfügung:

Kategorie	Objektname	Bemerkung
JSP	`pageContext`	Das `pageContext` Objekt der aktuellen Seite ermöglicht Zugriff auf alle impliziten JSP-Objekte
Scopes	`pageScope`	Eine `Map` für Zugriff auf die Attribute und Werte die im Scope von `page` gespeichert sind.
	`requestScope`	Eine `Map` für Zugriff auf die Attribute und Werte die im Scope von `request` gespeichert sind.
	`sessionScope`	Eine `Map` für Zugriff auf die Attribute und Werte die im Scope von `session` gespeichert sind.
	`applicationScope`	Eine `Map` für Zugriff auf die Attribute und Werte die im Scope von `application` gespeichert sind.
Request-Parameter	`param`	Eine `Map`, die die einzelnen Werte der bei dem Request übermittelten Parameter in `Strings` speichert. Entspricht dem Aufruf von `request.getParameter()`.
	`paramValues`	Eine `Map`, die alle Werte eines bei einem Request übermittelten Parameters in einem `String[]` speichert. Entspricht dem Aufruf von `request.getParameterValues()`.
Request-Header	`header`	Eine `Map`, die die einzelnen Werte der bei dem Request übermittelten Header in `Strings` speichert. Entspricht dem Aufruf von `request.getHeader()`.
	`headerValues`	Eine `Map`, die alle Werte eines bei einem Request übermittelten Header in einem `String[]` speichert. Entspricht dem Aufruf von `request.getHeaders()`.
Cookies	`cookie`	Eine `Map`, die alle bei dem Request übermittelten Cookies entsprechend ihrem Namen speichert. In Anlehung an `request.getCookies()`.
Initialisierung	`initParam`	Eine `Map`, die Zugriff auf die in `web.xml` mittels <context-param> definierten Initialisierungsparameter unter ihrem Namen ermöglicht: Entspricht dem Aufruf von `getServletContext().getInitParameter()`.

Tabelle 2-6
Implizite EL-Objekte

Function Tag Library

Wie schon bei der Einführung in die Programmierung von Tag-Libraries gezeigt, ist es mit Einführung der JSP-Spezifikation 2.0 möglich Funktionen zu implementieren. Die auf JSP 2.0 aufbauende JSTL Version 1.1 stellt zusätzlich zu den JSTL-Tags eine „Function Tag Library" zur Verfügung deren Funktionen vorwiegend zur Behandlung von Strings dienen. Sie ermöglichen

- die Umwandlung von Groß-/Kleinschreibung (`toLowerCase`, `toUpperCase`),
- die Lieferung von Substrings (`substring`, `substringAfter`, `substringBefore`),
- das Kürzen von Strings (`trim`),
- das Austauschen von Buchstaben eines Strings (`replace`),
- die Überprüfung ob ein String einen anderen enthält (`indexOf`, `startsWith`, `endsWith`, `contains`, `containsIgnoreCase`),
- das Aufsplitten eines Strings (`split`) in einen Array,
- das Umwandeln eines Arrays in einen String (`join`),
- die Umwandlung von Sonderzeichen (wie <, >, &, …), die als Tags interpretiert werden könnten, in für XML Dokumente gültige Zeichenfolgen (`<`, `>`, `&`, …) um XML-konforme Daten erstellen zu können (`escapeXml`).

Da für die hier besprochenen Themen nur die Core-Library-Funktionen benötigt werden, wird im Folgenden nur auf die Funktionen der Core-Bibliothek näher eingegangen. Mit der XML-Tag-Libary ließe sich zwar durch Verwendung von XML/XSL auch der View-Teil grundlegend beeinflussen, dies hat aber weniger mit dem MVC-Konzept als solches zu tun, und sollte eher in einem Vergleich mit XML-Frameworks wie Cocoon [33] [34] [35]behandelt werden.

Die Core-Tag-Library

Die Core-Tag-Library stellt „Kernfunktionen" zur Verfügung.

- Variablen Tags
- Schleifen
- bedingungsabhängige Programmflusssteuerung.
- Einfügen von Text
- Request-Umleitung

Um die JSTL-Tags in JSPs nutzen zu können müssen die Taglibraries installiert und in den JSPs deklariert werden.

Installation:
Zur Installation müssen die JAR-Dateien `jstl.jar` und `standard.jar` ins Verzeichnis `/WEB-INF/lib` der Web-Applikation kopiert werden. Die TLDs sind in den JAR-Dateien enthalten und müssen daher nicht ins `WEB-INF`-Verzeichnis kopiert werden. Einträge in `web.xml` sind nicht notwendig.

Deklaration

Bei der Deklaration gibt es folgendes zu beachten: da die EL in JSTL 1.0 ein optionaler Bestandteil war, existieren zwei verschiedene Arten der Deklaration pro Tag-Library, abhängig davon ob eine Run-Time-Auswertung der Attribute stattfinden soll oder nicht.

- Version mit EL (ohne Run-Time-Auswertung der übergebenen Werte) z.B. für die Core Bibliothek:
  `<%@ taglib prefix="c"  uri="http://java.sun.com/jsp/jstl/core" %>`
- Version ohne EL (mit Run-Time-Auswertung um dynamische Attribut-Werte mittels JSP-Ausdrücken zu realisieren) z.B. für die Core-Bibliothek:
  `<%@ taglib prefix="c_rt"  uri="http://java.sun.com/jsp/jstl/core_rt" %>`

Die JSTL 1.1 benötigt einen JSP-2.0-Container. Zu unangenehmen Seiteneffekten mit mysteriösen Fehlermeldungen kann es kommen, wenn man in `web.xml` die falsche DTD bzw. XML-Schema Definition verwendet.
Bei Tomcat 5 benötigt man die XML-Schema Version 2.4:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">

  ...

</web-app>
```

Variablen

Bei Variablen handelt es sich um in den Scopes abgespeicherte Objekte. Zum Zugriff auf Variablen existieren Tags zum Erstellen, Ausgeben und Löschen von (Objekt-)Variablen.
In Anlehnung an `<jsp:setProperty >` kann man mit `<c:set>` Variablen einen Wert zuweisen. Wenn die Variable im angegebenen scope schon existiert wird ihr der Wert zugewiesen. Existiert sie noch nicht, wird sie im angegeben scope erzeugt und mit dem Wert initialisiert.
`<c:set var="`*Name*`" scope="`*Scope*`" value="`*Ausdruck*`" />`

zur Ausgabe von Werten dient `<c:out >`. Dieser Tag hat kein Attribut über das Variablennamen übergeben werden könnten:
`<c:out value="`*Ausdruck*`" default="`*Ausdruck*`" escapeXml="`*Boolean*`">.`
Soll der Inhalt einer Variablen ausgegeben werden, dann muss die EL zu Hilfe genommen werden:
`<c:out value="${bsp.test}" default="Eigenschaft test ist null oder leer" >.`

Mit `<c:remove >` können Variablen wieder gelöscht werden.
`<c:remove var="`*Ausdruck*`" scope="`*Scope*`" >`

Schleifen

Es werden zwei Arten von Schleifen unterstütz. Die Iteration über einen Integer-Wertebereich oder über eine Collection. Beides ist mit <c:forEach> möglich.

Für eine Schleife über einen Werte-Bereich muss ein Star- (begin) und ein Endwert (end) angegeben werden, die anderen Attribute sind optional.

```
<c: forEach var="Name" varStatus="Name"
      begin="Ausdruck" end="Ausdruck" step="Ausdruck" >
  ...
</c:forEach>
```

Für eine Schleife über eine Collection muss die Collection dem Attribut items als EL-Ausdruck übergeben werden. Die anderen Attribute sind optional.

```
<c: forEach var="Name" items="Ausdruck" varStatus="Name"
      begin="Ausdruck" end="Ausdruck" step="Ausdruck" >
  ...
</c:forEach>
```

Das Attribut varStatus ist bei beiden Arten von Schleifen anwendbar. Hier kann ein Variablenname für eine Variable vom Typ javax.servlet.jsp.jstl.core.LoopTagStatus angegeben werden, über welche verschiedenen Eigenschaften vom Status der Schleife abgefragt werden können.

Eigenschaft	Rückgabewert	get-Methode	Beschreibung
begin	java.lang.Integer	getBegin()	Wert des begin-Attributes
count	int	getCount()	Wert des aktuellen Zählerstandes
current	java.lang.Object	getCurrent()	das aktuelle Element (einer Collection)
end	java.lang.Integer	getEnd()	Wert des end-Attributes
index	int	getIndex()	Wert des aktuellen Zählerstandes (beginnend mit Null)
step	java.lang.Integer	getStep()	Wert des Attributes Step
first	boolean	isFirst()	Zeigt an ob dies die erste Schleifenrunde ist
last	boolean	isLast()	Zeigt an ob dies die letzte Schleifen-runde ist

Tabelle 2-7
Scheifenstatus-Eigenschaften

Die core-Bibliothek stellt noch einen zweiten Schleifen-Tag bereit: <c:forTokens>. Dieser Tag entspricht der Java-StringTokenizer-Klasse. Die Syntax entspricht <c:forEach>. Zusätzlich muss über das Attribut delims das zur Trennung der einzelnen Tokens verwendete Trennsymbol angegeben werden

```
<c: forEach var="Name" items="Ausdruck" delims="Ausdruck" varStatus="Name"
      begin="Ausdruck" end="Ausdruck" step="Ausdruck" >
  ...
</c:forEach>
```

Bedingungsabhängige Programmflusssteuerung

Zur Auswertung von Bedingungen und Verzweigung sind zwei Tag-Konstrukte verfügbar.
Der <c:if>-Tag wertet einen mittels test-Attribut übergebenen Ausdruck aus. Wenn die Auswertung der Bedingung "true" ergibt, wird der Rumpf des Tags abgearbeitet. Wenn nicht, wird der Rumpf ignoriert. Das Ergebnis der Auswertung kann in einer Variablen (var) gespeichert werden.

```
<c:if test="Ausdruck" var="Name" scope="Scope" >
  ...
</c:if>
```

Das Tag-Konstrukt `<c:choose>-<c:when>-<c:otherwise>` ermöglich die Auswertung von "mutually exclusive" Bedingungen.

```
<c:choose>
  <c:when test="Ausdruck" >
    ...
  </c:when>

  [optional weitere<c: when>- Tags mit weiteren test-Bedingungen]

  <c:otherwise>
    ...
  </c:otherwise>
</c:choose>
```

Einfügen von Text

Zum Einfügen sind in der JSP-Spezifikation zwei Methoden definiert:

- die include-Direktive `<%@ include file="..." %>`
- die include-Aktion `<jsp:include page="..." />`.

Beim `<c:import>`–Tag der JSTL handelt es sich um eine verbesserte Version der include Aktion. Wie `<jsp:include />` wird auch `<c:import />` zur Ausführungszeit ausgewertet.

```
<c:import url="Ausdruck" context="Ausdruck" charEncoding="Ausdruck"
                var="Name" scope="Scope" >
  <c:param name="Ausdruck" value="Ausdruck">
  [optional weitere<c: param>- Tags]
</c:import>
```

Das `<c:import>` - Tag ist nicht auf lokale Dateien beschränkt. Das URL-Attribut akzeptiert komplette URIs inklusive Übertragungsprotokoll. Jedes in `java.net.URL` angeführte Protokoll wird unterstützt. Mit Hilfe des `context`-Attributes kann man über einen relativen URL, der sich nicht im eigenen Context befindet, auf andere Quellen zugegriffen werden.
Der einzufügende Text kann auch in der unter `var` angegeben Variablen abgespeichert werden um dann z.B. mit Hilfe der JSTL-XML-Library vor der Ausgabe weiter bearbeitet zu werden.

Request-Umleitung

Wie bei der Erklärung der Weiterleitung mittels RequestDispatcher-Mechanismus in der Servlet-Einführung schon erwähnt, besteht ein Unterschied zwischen der *Weiterleitung* (=*Forward*) einerseits und der *Umleitung* (= *Redirect*) andererseits.
Um Redirects zu ermöglichen bietet die JSTL den Tag <c:redirect>:

```
<c:redirect url="Ausdruck" context="Ausdruck" >
  <c:param name="Ausdruck" value="Ausdruck">
  [optional weitere<c: param>- Tags]
</c:redirect>
```

Die Weiterleitung geschieht für Benutzer und Client unsichtbar, da sich der URL nicht verändert. Bei einem Redirect hingegen erhält der Client vom Webserver die Aufforderung eine andere Web-Ressource zu laden. Für den Benutzer ist das insofern von Bedeutung dass der die Adresse der neuen Adresse als Bookmark abspeichern kann.

MVC mit JSP und JSTL

Mit Hilfe der JSTL und der EL ist es nun fast zur Gänze möglich das obige MVC Beispiel so umzuschreiben, dass es keinen Scriptlet-Code mehr enthält.

```
<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" />

<jsp:setProperty name="bsp" property="*" />

<html>
  <head>
    <title>Input - Session</title>
  </head>
  <body>
    Input - Session<br />

    <form action="jstl_controller.jsp" method="get" >
      Zahl: <input type="text" name="zahl" value="${bsp.zahl}" />
      <br />
      Test: <input type="text" name="test" value="${bsp.test}" />
      <br />
      <input type="submit" />
    </form>

  <body>
</html>
```

Beispiel 2-60
`input_view.jsp`

```
<%@ taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>

<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" />

<jsp:setProperty name="bsp" property="*" />

<c:choose>
  <c:when test="${ param.test eq null || param.test eq '' }" >
    <jsp:forward page="jstl_input_view.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="jstl_output_view.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 2-61
`controller.jsp`

```
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>

<jsp:useBean id="bsp" class="at.ac.akhwien.mvc.model.BeispielBean"
                      scope="session" />

<html>
  <head>
    <title>Output</title>
  </head>
  <body>
    <p>
```

```
      zahl: ${bsp.zahl} <br />
      test: ${bsp.test} <br />
    </p>
    <hr />
    Zur&uuml;ck zum <a href="jstl_input_view.jsp">Start</a>
  <body>
</html>

<sess:invalidate/>
```

Beispiel 2-62
`output_view.jsp`

Ganz ohne zusätzliche Taglibraries kommt das Beispiel jedoch nicht aus. In `output_view.jsp` wird zusätzlich noch die Session-Tag-Library des Apache Taglib-Projekts verwendet, um die Session ungültig zu machen.

Da die anderen Funktionen der JSTL für das Verständnis der hier behandelten Thematik nicht von Bedeutung sind sei an dieser Stelle auf andere Quellen verwiesen. Eine sehr gute weiterführende Einführung in die Verwendung der JSTL ist auf den DeveloperWorks-Seiten von IBM zu finden[36]. Auch die JSTL-Spezifikation[37] leistet gute Dienste beim Nachschlagen.

3. Methodik

Nachdem vorgestellt wurde, wie sich mit Servlets und JSPs eine MVC-Web-Anwendung realisieren lässt, und weiters die Möglichkeiten gezeigt wurden Programm-Code in einen Controller und in benutzerdefinierte Tag-Libraries auszulagern, soll nun anhand einer Aufgabenstellung für eine einfache Web-Anwendung untersucht werden, wo die Unterschiede in der Implementation einer Web-Anwendung bei Verwendung von JSP/JSTL, dem MVC-Framework Struts und JavaServer Faces liegen.

Folgende Fragen sollen hierbei behandelt werden:

- Ist eine vollständige Trennung von HTML und Java-Code erreichbar?
- Verbessern sich die Lesbarkeit und damit die Wartbarkeit des Codes?
- Gibt es Einschränkungen in View-Design?
- Gibt es Einschränkungen bei der Bedienbarkeit?
- Gibt es einen merkbaren Einfluss auf die Performance?

Um die hier abgedruckten Programme zu testen benötigt man keine kommerzielle Entwicklungsumgebung. Der Programmcodes sind so einfach wie möglich gehalten und kann mit einem einfachen Texteditor erstellt werden. Im Rahmen der Vorstellung der Implementation sind nur wesentlichen Teile des Codes abgedruckt. Der vollständige Quellcode befindet sich im Anhang.
Die Datei mit den Aufgabestellungen ist nicht aufgeführt, da es sich um Originalfragen des Instituts für Anatomie der Medizinischen Universität Wien handelt. Das Dateiformat ist sehr einfach gehalten und im Anhang beschrieben.

Zum Compilieren wurde das Sun JDK 1.4.2 verwendet. Als Built-Tool sei Ant oder Maven empfohlen. Als Servlet-Engine diente Tomcat 5.0. Weitere benötigte Komponenten sind die Apache Taglibs-JSTL Version 1.1, Struts 1.2.7 und JSF Referenz Implementation von Sun Version 1.0.

3.1. Aufgabenstellung:

Die Web-Anwendung beruht auf einer vorgegebenen Aufgabenstellung. Im Jahr 1996 wurde am Institut für medizinische Computerwissenschaften in Zusammenarbeit mit dem Institut für medizinische Aus- und Weiterbildung am AKH-Wien eine Web-Anwendung zur Prüfungsvorbereitung, das Knochenkolloquium ("Knoko"), entwickelt. Dabei handelte es sich um in der Programmiersprache C geschriebene CGI -Programme, die all die eingangs schon erwähnten Probleme in der Wartung boten. Der HTML-Code war in printf-Anweisungen enthalten wodurch der C-Programmcode und HTML untrennbar durchmischt waren. Nach einem Umzug auf einen anderen Webserver funktionierte die Anwendung nicht mehr, da absolute Links in den Programmcode integriert waren. Während dies noch mit einer trickreichen Serverkonfiguration zu lösen war, machten sich die Unterschiede in der Interpretation von HTML durch verschiedene Browser vor allem bei der Umlautdarstellung unangenehm bemerkbar. Da die Programme in C geschrieben waren, war nach jeder Änderung eine Neukompilierung notwendig.

Der Benutzer soll über einen Webbrowser ein Lernprogramm aufrufen können, das dem Benutzer nacheinander mehrere Fragen stellt. Das Programm generiert einen Test, der beim ersten Aufruf erzeugt wird. Hierbei handelt es sich um eine Untermenge von Multiple-

Choice-Fragen die zufällig vom Programm aus einem Fragenpool entnommen werden. Der Fragenpool inklusive der Antwortmöglichkeiten und der richtigen Lösung befindet sich in einer Textdatei.
Der Programm befindet sich in einem „Lernmodus“, d.h. der Benutzer bekommt jeweils immer nur eine Aufgabenstellung und je nach Fragentyp mehrere Antworten oder ein Bild präsentiert. Es existieren 3 verschiedene Multiple-Choice-Fragentypen, die auf den Empfehlungen des Institut für Aus-, Weiter- und Fortbildung (IAWF) der Universität Bern, bzw. des US-amerikanischen National Board of Medical Examiners (NBME) basieren:

- Einfachauswahl (A) - die Aufgabe beinhaltet eine Aufgabenstellung und 5 Wahlantworten (oder ein Bild) die angekreuzt bzw. angeklickt werden können. Es ist nur eine Antwort richtig.
- Antwortkombinationsaufgabe (Kprim) - auf eine Frage oder unvollständige Aussage folgen 3 Wahlantworten, bezeichnet mit 1, 2 und 3. Zunächst ist für jede dieser 3 Antworten voneinander unabhängig zu entscheiden, ob sie richtig oder falsch ist. Danach ist die entsprechende der 5 vorgegebenen Antwortkombinationen (A-E) zu wählen. Der Kombinationsschlüssel bleibt für die gesamte Prüfung unverändert gleich und wird bei jeder Aufgabe wiederholt angeführt: A(1+2+3) B(1+2) C(2+3) D(1) E(3).
- Zuordnungsfrage (B) - Bei diesem Typ werden dem Prüfling eine oder mehrere Aufgabenstellungen zusammen mit 5 Wahlantworten (mit den Bezeichnern A-E etikettiert) oder einem Bild präsentiert. Es ist nur 1 Wahlantwort (A-E) richtig. (Im Rahmen des Lernprogramms wird dem Benutzer immer nur eine Aufgabenstellung mit 5 Wahlantworten gezeigt).

Weiters soll angeführt sein: Dauer des Tests seit Beginn, Anzahl der gestellten und richtigen Fragen. Durch Anklicken einer 1-aus-N-Auswahl soll der Benutzer zuerst seine Auswahl treffen und auch wieder ändern können Wenn er sich für eine Wahlantwort entschieden hat, soll er die Auswahl durch Abschicken des Formulars bestätigen.
Wenn der Benutzer die Frage richtig beantwortet hat, wird ihm das sofort mitgeteilt. Zusätzlich bekommt er noch ergänzende Informationen präsentiert. Wenn der Benutzer die Frage noch einmal gestellt bekommen will, so kann er das dem Programm übermitteln.
Falls die Frage falsch beantwortet wurde, wird dies dem Benutzer auch sofort mitgeteilt. Es werden ihm die richtige Lösung sowie zusätzliche Informationen präsentiert. Ob die Aufgabe noch einmal präsentiert werden soll ist bei falschen Antworten als Voreinstellung aktiviert, kann aber vom Benutzer deaktiviert werden.
Um die nächste Frage gestellt zu bekommen muss der Benutzer diese durch Anklicken eines Buttons anfordern. Er kann aber auch den Test vorzeitig beenden.
Wenn alle Fragen des Tests und alle Wiederholungsfragen gestellt sind oder der Test vorzeitig beendet wurde, wird dem Benutzer eine Statistik präsentiert. Hier wird ihm mitgeteilt wie viele Fragen gestellt wurden, wie viele richtig beantwortet wurden, wie lange er vom Start des Tests gebraucht hat, wie viele Fragen er pro Minute beantwortet hat.

3.2. Entwurf

Model

Alle drei MVC-Varianten sollen das gleiche Model nutzen. Es verwaltet die Fragensammlung, den Testverlauf und die Aufgaben und muss folgende Funktionen zur Verfügung stellen:

- die Fragensammlung einlesen,
- Tests generieren,
- zu stellenden Aufgaben liefern,

- die Antworten auf Korrektheit zu überprüfen,
- die Zeit seit Beginn der Testsitzung messen,
- eine Statistik führen (Anzahl der gestellten, verbleibenden und richtigen Antworten) und
- dafür zu sorgen, dass zu wiederholende Fragen erneut gestellt werden.

Es existieren 3 Klassen: Aufgabe, Aufgabenkatalog und Testverlauf. Die drei Klassen stammen alle von `java.lang.Object` ab und sind von der Servlet/JSP-API unabhängig. Sie könnten demnach anstatt in einer Web-Applikation auch zur Programmierung einer eigenständigen Java-Applikation verwendet werden.

Je eine Frage ist ein je einem Aufgabenobjekt gespeichert. Die Klasse Aufgabe enthält die notwendigen get- und set-Methoden um auf ihre Eigenschaften wie Frage, Wahlantworten, Lösung usw. zugreifen zu können.

Das Objekt Aufgabenkatalog enthält die Informationen über die Aufgaben in einer Liste von Aufgaben-Objekten. Die Aufgaben müssen aus einer Textdatei gelesen werden. Der Dateiname dieser Datei wird dem Objekt über eine Methode mitgeteilt, da die Konstruktoren bei Beans per Definition keine Parameter enthalten dürfen.

Das Objekt Aufgabenkatalog enthält Daten, die der gesamten Web-Applikation von Anfang an zur Verfügung stehen müssen. Es ist daher sinnvoll das Objekt beim Start der Web-Applikation im application-Objekt zu speichern.

Ein einzelner Test wird generiert und verwaltet vom Testverlauf-Objekt. Damit auf die gespeicherte Fragensammlung zugegriffen werden kann, existiert eine Methode die eine Referenz auf das Objekt Aufgabenkatalog speichert – siehe unten. Das Objekt enthält Methoden die die Controller entlasten sollen. Es ermöglicht stets den Zugriff auf die aktuelle Aufgabe und wird über eine get-Methode geliefert. Soll eine neue Aufgabe geliefert werden, muss man das Objekt auffordern die nächste Aufgabe bereit zu stellen. Rückgabewert ist "neu", wenn eine neue Aufgabe verfügbar ist, oder "ende" wenn der Test zu Ende ist.
Dem Objekt wird die Antwort des Benutzers über eine set-Methode übergeben. Ob die Antwort korrekt beantwortet wurde wird von einer Methode überprüft und mit den Rückgabewerten "richtig" und "falsch" beantwortet.

Das Objekt Testverlauf ist ein Objekt das nur für einen Benutzer für die Dauer eines Tests gültig ist. Es wird daher in im session-Objekt abgespeichert. Damit Testverlauf auf Aufgabenkatalog zugreifen kann muss eine Referenz auf Aufgabenkatalog übergeben werden. Ebenso benötigt Testverlauf die Angabe wie viele Fragen pro Test gestellt werden sollen. Auch diese Information ist in web.xml als `<context-param>` hinterlegt.

Controller

Die Controller-Funktionalität kann auf den Möglichkeiten, die das Model bietet aufbauen. Es muss gewährleistet sein, dass auf die Model-Beans zugegriffen werden kann bzw. dass die Model Beans erstellt werden. Das Flussdiagramm zeigt wie die Controller abhängig von den Benutzerantworten auf andere Seiten verzweigen müssen.

Während bei JSP/JSTL die Controller als JSPs implementiert werden, bedienen sich Struts und JSF der Hilfe von Servlets und in Konfigurationsdateien ausgelagerte Navigationsregeln. Die Verzweigungen auf andere Seiten sind beim JSP/JSTL-Ansatz in die `<form>`-Tags eincodiert. Struts hingegen verwendet die Konfigurationsdatei `struts-config.xml` und JSF

die Konfigurationsdatei `faces-config.xml`, die sich beide in `/WEB-INF/` befinden. In diesen Dateien werden unter anderem auch Beans deklariert.

Das Flussdiagramm zeigt den Ablauf. Nachdem der Benutzer die Eingangsseite aufgerufen hat, muss ein Testverlauf-Objekt im Session-Scope angelegt werden.
Mit der Anforderung für eine neue Aufgabe wird vom Objekt Testverlauf bestätigt, dass es sich um eine neue Aufgabenstellung handelt, in dem der Wert "neu" zurückgeliefert wird. Dies geschieht so lange bis keine neue Aufgabe mehr geliefert werden kann, was mit "ende" beantwortet wird. Steht eine neue Aufgabenstellung so wird auf das entsprechende View verzweigt, das die Frage mit den Wahlantworten darstellt. Gibt es keine neue Aufgabenstellung mehr, dann wird auf das View geleitet, das die Test-Statistik anzeigt – von hier geht es nur wieder zum Start.
Wenn der Benutzer die Frage beantwortet hat, kann der Controller die Antwort an die Testverlauf-Bean übergeben und mittels einer Methode die Aufgabe korrigieren lassen. Liefert die Methode den Wert "richtig", dann wird ein View angezeigt, das dem Benutzer mitteilt, dass seine Beantwortung richtig war und er hat die Möglichkeit anzugeben, ob er die Frage noch einmal gestellt bekommen soll. Ist der Rückgabewert "falsch" wird auf ein View umgeleitet, das mitteilt, dass die Antwort falsch war. Auch hier kann der Benutzer auswählen ob die Frage noch einmal gestellt werden soll.
Wenn der Benutzer die nächste Frage anfordert, wird in jedem Fall wieder auf den ersten Controller verwiesen, der zuerst dafür sorgt, dass die Frage erneut gestellt wird, wenn dies vom Benutzer gewünscht war. Danach wird wieder eine neue Aufgabe von Testverlauf angefordert.

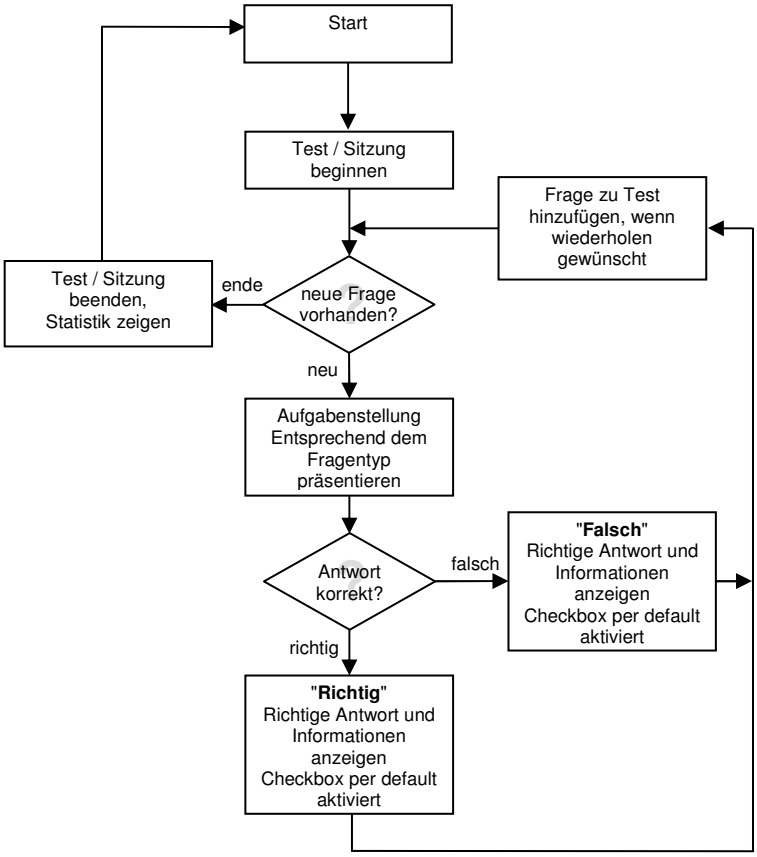

Abbildung 3-1
Flussdiagramm für Controller-Funktionen

Views

Die Aufgabenstellung und die Antwortmöglichkeiten werden in Abhängigkeit vom Fragentyp im entsprechenden Layout dargestellt und mittels Radio-Buttons eine Auswahlmöglichkeit angeboten.

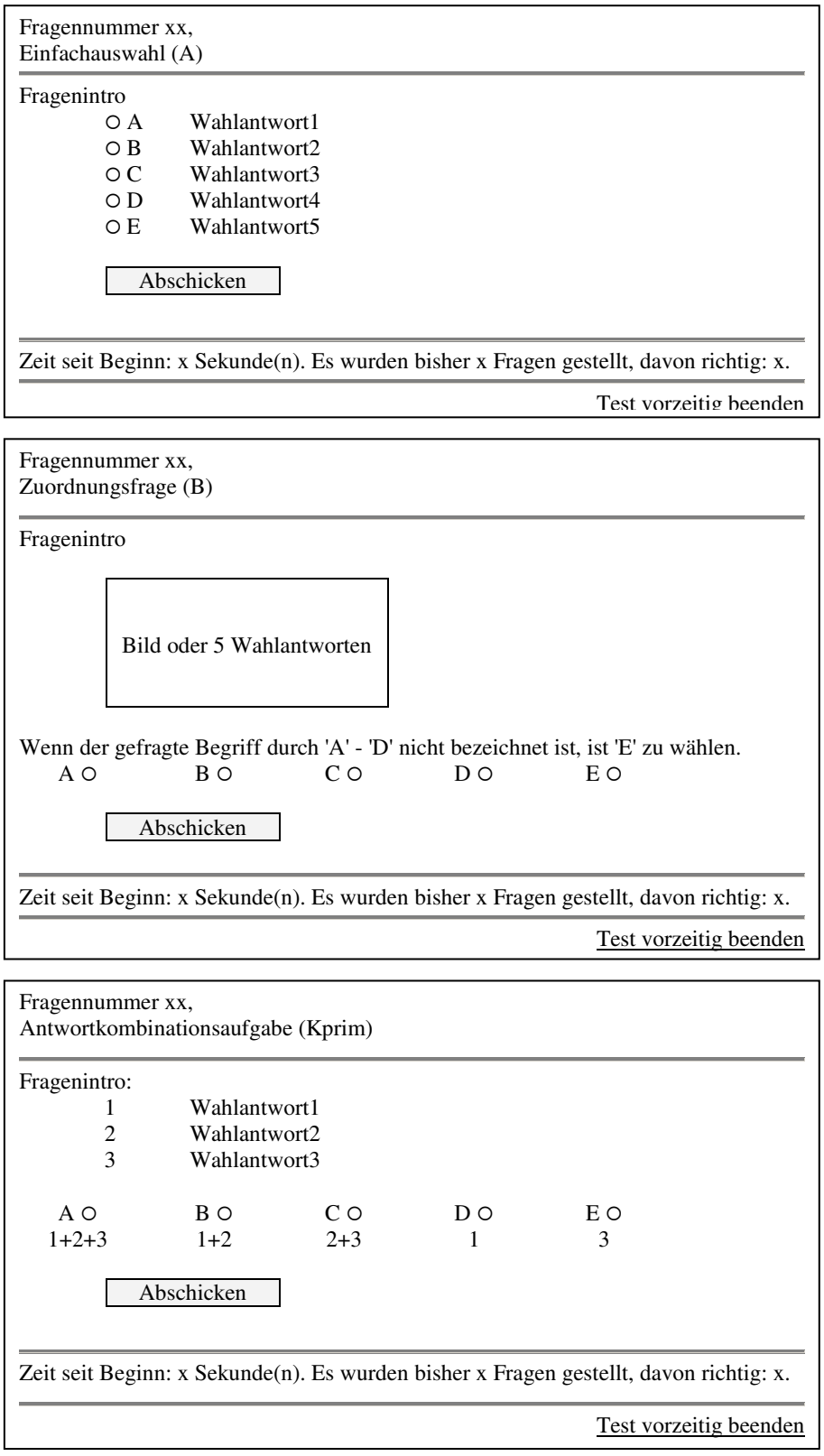

Abbildung 3-2
Layouts der3 Fragentypen

Auf den Feedbackseiten wird nach der Korrektur das Ergebnis der Überprüfung mitgeteilt. Weiters kann der Benutzer über anwählen oder abwählen einer Checkbox mitteilen, ob die gerade gestellte Frage nochmals gestellt werden soll. War die Antwort richtig, ist die Checkbox per Voreinstellung nicht angewählt. War die Antwort falsch, so ist die Checkbox per Default angekreuzt.

Die Statistikseite am Ende der Prüfung zeigt an wie viele Fragen insgesamt gestellt und wie viele davon richtig beantwortet wurden, wie viel Zeit seit Start des Tests vergangen ist und wie viele Fragen pro Minute beantwortet wurden.

4. Ergebnisse

Hier werden der Quellcode der drei funktionsfähigen Implementierungen und der gemeinsamen Komponenten auszugsweise aufgelistet und kommentiert. Nach den Quellcode-Listings der Views sind die Screenshots dieser Views abgebildet. Neben den Unterschieden in der Darstellung dokumentieren die Bilder auch die verschiedenen Aufrufe in der Adresszeile.

4.1. Implementation gemeinsamer Komponenten

Model

Alle drei MVC-Varianten verwenden diese 3 JavaBeans als Model welches in Form von drei JavaBeans implementiert ist:

```
at.ac.akhwien.mvc.model.Aufgabe
at.ac.akhwien.mvc.model.Aufgabenkatalog
at.ac.akhwien.mvc.model.Testverlauf
```

Die Klasse `Aufgabe` enthält die notwendigen get- und set-Methoden um auf ihre Eigenschaften zugreifen zu können: `getFragenintro()`, `getWahlantwort1()`, `getWahlantwort2()`, ... , `getLoesung()` usw. Zusätzlich kann mit `setFragestellen()` bestimmt werden, ob eine Frage gestellt werden soll. Mit Hilfe der Methode `kopiereProps()` können die Eigenschaften in ein anderes Objekt vom Typ `Aufgabe` kopiert werden.

Die Klasse `Aufgabenkatalog` enthält die Informationen über die Aufgaben in einer Liste von `Aufgaben`-Objekten. Die Aufgaben müssen aus einer Textdatei gelesen werden. Der Dateiname dieser Datei wird über die Methode `setDateiname()` mitgeteilt. Beim ersten Aufruf der Methode `getAufgabe()` wird die Datei geladen und die Liste von Aufgaben erstellt und es wird die Aufgabe entsprechend der übermittelten Aufgabennummer zurückgeliefert.

Ein einzelner Test wird generiert und verwaltet von `Testverlauf`. Um auf die Fragensammlung zugreifen zu können wird der Methode `setKatalog()` eine Referenz auf das `Aufgabenkatalog`-Objekt übergeben. Mit `setTestumfang()` kann bestimmt werden, wie viele verschiedene Aufgaben ein Test enthalten soll. Beim ersten Aufruf der Methode `neueAufgabe()` wird der Test generiert. Der Rückgabewert kann "`neu`" oder "`ende`" sein. Der Wert "`neu`" bedeutet, dass ein neues Aufgabenobjekt erstellt wurde. Dieses stellt die Methode `getAufgabe()` bereit. Der Rückgabewert "`ende`" bedeutet, dass alle Aufgaben des Tests gestellt wurden. Zur Übermittlung der Antwort des Benutzers dient `setAntwort()`. Ob die

Antwort korrekt war beantwortet die Methode `aufgabekorrigieren()` mit den Rückgabewerten "`richtig`" und "`falsch`".

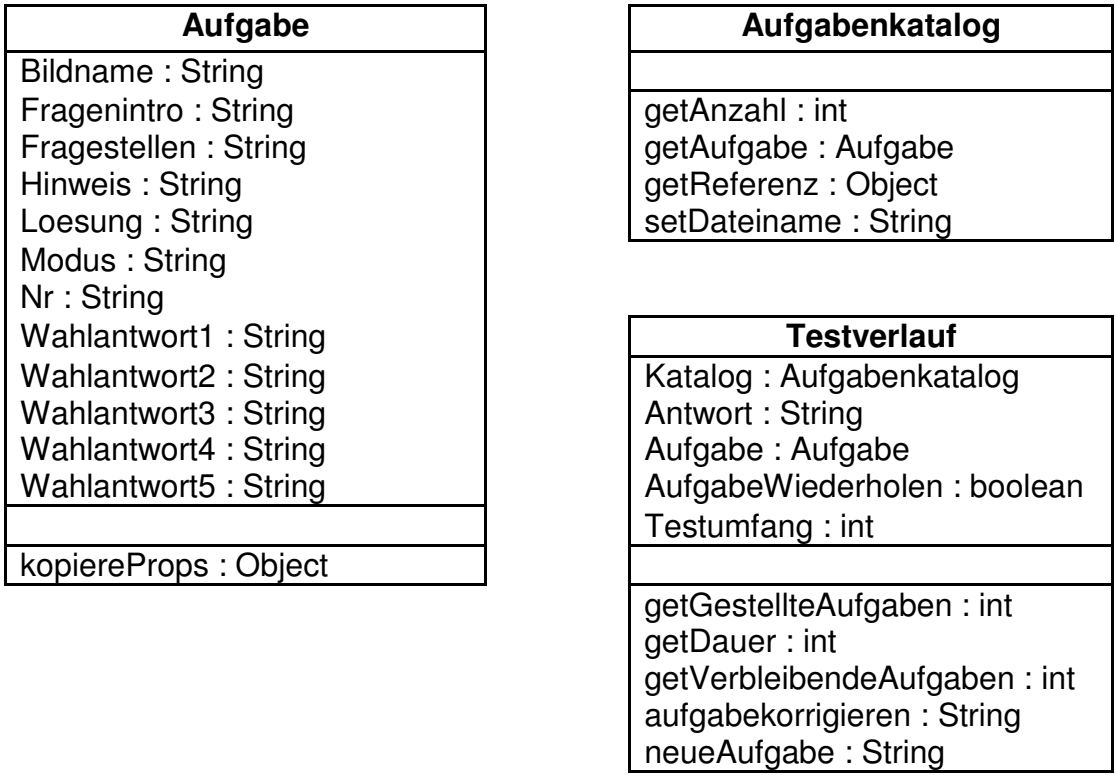

Abbildung 4-1
Klassendiagramme der Model-Beans

Eine Genaue Übersicht über die Methoden inklusive Beschreibung befindet sich im Anhang.

Das Objekt `Aufgabenkatalog` soll beim Start der Web-Applikation im `application`-Objekt gespeichert werden. Ein Problem dabei ist, dass `Aufgabenkatalog` der Dateiname übergeben werden muss – und zwar mit kompletter Pfadangabe. Diese Aufgabe übernimmt die Methode `ServletContext.getRealPath()`, mit der relativen Pfadangabe als Parameter. Damit dies unmittelbar bei Start geschieht bieten sich zwei Möglichkeiten an dies programmtechnisch zu realisieren:

- als Listener, der darauf "horcht" wann der `ServletContext` initialisiert wird,
- als Servlet, dass beim Start der Web-Applikation startet.

Hier wird die Methode `contextInitialized()` von `ServletContextListener` implementiert (`at.ac.akhwien.mvc.model.SCInit`). Erst wenn diese Methode abgearbeitet ist nimmt der Webserver Requests für die Web-Applikation entgegen. Die Methode holt sich zuerst mit `getInitParameter()` den in `web.xml` unter "`rel_dateiname`" gespeicherten Dateinamen inklusive relativem Pfad (in Bezug auf die Web-Applikation). Anschließend wird mittels `getRealPath()` der absolute Pfad ermittelt und der volle Dateinamen als Objekt im `ServletContext`-Objekt (=`application`) gespeichert. Auf dieses kann von Servlets oder JSP leicht zugegriffen werden. Damit der Listener beim Start gestartet wird, muss er in `web.xml` deklariert sein.

Web Application Deployment Discriptor `web.xml`

Im Web Application Deployment Discriptor `web.xml` werden bei allen Versionen des Lernprogramms die Initialisierungsparameter "`rel_dateiname`" und "`testumfang`" definiert und der Listener `SCInit` deklariert. Weiters werden per `<welcome-file-list>` die Dateien definiert, die zurückgeliefert werden sollen wenn im Client-Request nur ein Verzeichnis ohne Dateinamen angefordert wird.

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">


  <display-name>Knoko Pruefungsvorbereitung</display-name>
  <description>
    Diese Webapplication dient zur Pruefungsvorbereitung fuer das
    Knochenkolloquium.
  </description>

  <context-param>
    <param-name>rel_dateiname</param-name>
    <param-value>/WEB-INF/knokofragen.txt</param-value>
  </context-param>

  <context-param>
    <param-name>testumfang</param-name>
    <param-value>6</param-value>
  </context-param>

  <listener>
    <listener-class>at.ac.akhwien.mvc.model.SCInit</listener-class>
  </listener>

  <taglib>
    <taglib-uri>http://akh-wien.ac.at/functionwrapper</taglib-uri>
    <taglib-location>/WEB-INF/FunctionWrapper.tld </taglib-location>
  </taglib>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  ...

</web-app>
```

Beispiel 4-1
`web.xml`

4.2. Implementation der drei MVC-Beispiele

Controller und Views in der JSP/JSTL-Version

Die JSP/JSTL-Version setzt JSP 2.0 und JSTL 1.1 voraus, da sie die Möglichkeiten der JSP-EL und Funktionen nutzt.

Installation

Wie schon oben ausführlich beschrieben sind nur die JAR-Dateien `jstl.jar` und `standard.jar` ins Verzeichnis `/WEB-INF/lib` der Web-Applikation zu kopieren und die benötigte Bibliothek in den betreffenden JSPs mittels `<%@ taglib>`-Direktive zu deklarieren – Details siehe im Kapitel JSTL.

Funktionsweise

Die JSP/JSTL-Version beruht auf dem oben schon vorgestellten MVC-Konzept für JSPs: Controller-JSPs sind rein für Zugriff auf das Model und das Weiterleiten auf View-JSPs zuständig.

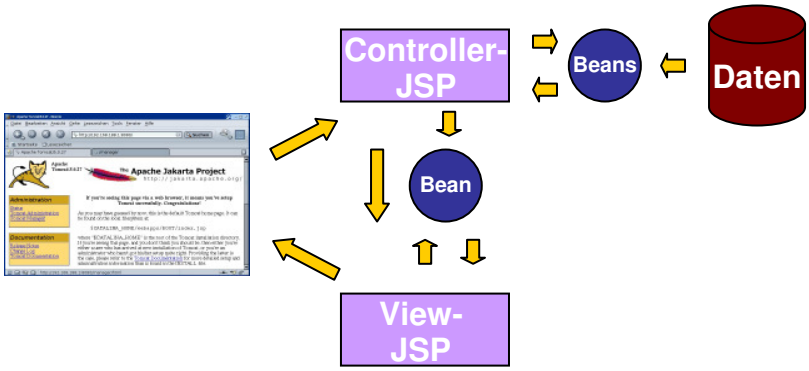

Abbildung 4-2
MVC mittels JSP mit Zugriff auf Daten

Für den Zugriff auf das Modell werden hier die mit JSP 2.0 eingeführten Funktionen genützt. Dazu muss der zugehörige Tag-Library-Deskriptor in `web.xml` deklariert werden.

```
<taglib>
  <taglib-uri>http://akh-wien.ac.at/functionwrapper</taglib-uri>
  <taglib-location>/WEB-INF/FunctionWrapper.tld </taglib-location>
</taglib>
```

Beispiel 4-2
Ausschnitt aus `web.xml`

Der Tag-Library-Deskriptor `FunctionWrapper.tld` definiert den Zugriff auf zwei Funktionen der Klasse `at.ac.akhwien.mvc.taglibs.Functionwrapper`: `neueAufgabe` und `aufgabekorrigieren`

```
  <function>
    <name>neueAufgabe</name>
    <function-class>
      at.ac.akhwien.mvc.taglibs.FunctionWrapper
    </function-class>
    <function-signature>
      java.lang.String neueAufgabe(at.ac.akhwien.mvc.model.Testverlauf)
    </function-signature>
  </function>

  <function>
    <name>aufgabekorrigieren</name>
    <function-class>
      at.ac.akhwien.mvc.taglibs.FunctionWrapper
    </function-class>
    <function-signature>
      java.lang.String aufgabekorrigieren(at.ac.akhwien.mvc.model.Testverlauf)
    </function-signature>
  </function>
```

Beispiel 4-3
Ausschnitt aus `FunctionWrapper.tld`

Die Klasse `FunctionWrapper` dient als "Wrapper" für den Zugriff auf die Methoden `neueAufgabe()` und `aufgabekorrigieren()` der Klasse `Testverlauf`.

```
public class FunctionWrapper
{
  public static String neueAufgabe(Testverlauf tvObject)
  {
    return tvObject.neueAufgabe();
  }

  public static String aufgabekorrigieren(Testverlauf tvObject)
  {
    return tvObject.aufgabekorrigieren();
  }
```

Beispiel 4-4
Ausschnitt aus `FunctionWrapper.java`

Die Homepage des Lernprogramms nimmt nicht an Sessions teil und verweist auf den Start des Lernprogramms, die Controller-JSP `aufgabe.jsp`.

```
<%@page  session="false" %>

<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabe.jsp">Start</a>.

    </hr>

  </body>
</html>
```

Beispiel 4-5
`index.jsp`

Die Controller-JSP `aufgabe.jsp` deklariert, dass sie auf das `Aufgabenkatalog`-Objekt zugreifen will. Ist dieses noch nicht im Scope `application` vorhanden, wird es automatisch dort erstellt und danach sofort die Methode `setDateiname()` mit dem Dateinamen aufgerufen, den der Listener `SCInit` im `application`-Scope unter "`dateiname`" abgelegt hat.
Weiters wird vereinbart, dass die Seite auf das `session`-Objekt `Testverlauf` zuzugreifen will. Falls noch nicht vorhanden, wird eines im Scope `session` angelegt und sofort danach die Methode `setKatalog()` mit einer Referenz auf das Objekt `Aufgabenkatalog` und die Methode `setTestumfang` mit dem in `web.xml` definierten Initialisierungswert `testumfang` aufgerufen.
Wurde die Seite von den Seiten `richtig.jsp` oder `falsch.jsp` aus aufgerufen übergeben diese den Parameter ob eine Aufgabe noch einmal gestellt werden soll. Falls also der Parameter "`aufgabeWiederholen`" übergeben wurde, wird sein Wert der Eigenschaft `aufgabeWiederholen` zugewiesen.
Danach wird über einen Funktionsaufruf eine neue Aufgabe angefordert. Der Rückgabewert der Funktion wird in der Variablen `neueaufgabe` gespeichert und anschließend ausgewertet. Kann eine neue Aufgabe geliefert werden ("`neu`"), dann verzweigt der Controller auf `aufgabenstellung.jsp`. Wenn keine Aufgabe mehr vorhanden ist ("`ende`"), wird auf `ende.jsp` weitergeleitet.

```
%@page session="true" %>

<%@ taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fw"   uri="http://akh-wien.ac.at/functionwrapper" %>

<jsp:useBean id="Aufgabenkatalog"
     class="at.ac.akhwien.mvc.model.Aufgabenkatalog" scope="application">

  <jsp:setProperty name="Aufgabenkatalog"
                   property="dateiname" value="${applicationScope.dateiname}" />
</jsp:useBean>

<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session">
  <jsp:setProperty name="Testverlauf"
                   property="katalog" value="${Aufgabenkatalog.referenz}" />
  <jsp:setProperty name="Testverlauf"
                   property="testumfang" value="${initParam.testumfang}" />
</jsp:useBean>


<jsp:setProperty name="Testverlauf" property="aufgabeWiederholen" />

<c:set var="neueaufgabe" value="${fw:neueAufgabe(Testverlauf)}" />

<c:choose>
  <c:when test="${ neueaufgabe eq 'neu'}" >
    <jsp:forward page="aufgabenstellung.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="ende.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 4-6
Controller-JSP `aufgabe.jsp`

Die JSP `aufgabenstellung.jsp` ist für das Layout der verschieden Fragentypen zuständig. Zu Begin wird überprüft, ob das Objekt Testverlauf im session-Scope vorhanden ist. Ist dies nicht der Fall, dann wird die aktive session beendet und an die Startseite weitergeleitet.
Die Informationen der aktuellen Aufgabe werden über rekursive EL-Ausdrücke ausgeben: z.B. entspricht `${Testverlauf.aufgabe.nr}` dem Scriptlet
`<%= Testverlauf.getAufgabe().getNr() %>`
Entsprechend dem Fragetyp (`modus`) wird das entsprechende Layout ausgegeben. Ist eine Bilddatei vorhanden wird diese ausgegeben (betrifft nur Fragetypen A und B).
Die Controller-JSP `korrektur.jsp` wird mit dem Parameter `antwort` aufgerufen wenn der Benutzer den `Submit`-Button anklickt.

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>

<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>

<html>

    ...

  <body>

    ...

    Fragennummer  ${Testverlauf.aufgabe.nr},
    <form method="post" action="korrektur.jsp">

      <c:choose>
        <c:when test="${Testverlauf.aufgabe.modus==1}" >
          Einfachauswahl (A)

            ${Testverlauf.aufgabe.fragenintro}
            <c:choose>
              <c:when test="${Testverlauf.aufgabe.bildname!=''}" >
                <img src="${Testverlauf.aufgabe.bildname}" />
              </c:when>
            </c:choose>
              <label>
                <input type="radio" name="antwort" value="A">
                A  ${Testverlauf.aufgabe.wahlantwort1}
              </label>

              <label>
                 <input type="radio" name="antwort" value="B">
                 B  ${Testverlauf.aufgabe.wahlantwort2}
              </label>

              ...

        </c:when>

        <c:when test="${Testverlauf.aufgabe.modus==2}" >
          Antwortkombinationsaufgabe (Kprim)

          ...

        </c:when>

        <c:when test="${Testverlauf.aufgabe.modus==3}" >
          Zuordnungsfrage (B)
```

```
          ...

        </c:when>
      </c:choose>

      <input type="submit" value="Antwort abschicken">
    </form>

    Zeit seit Beginn: ${Testverlauf.dauer} Sekunde(n).
    Es wurden bisher ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    davon richtig: ${Testverlauf.richtigeAufgaben}.

    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 4-7
Ausschnitt aus `aufgabenstellung.jsp`

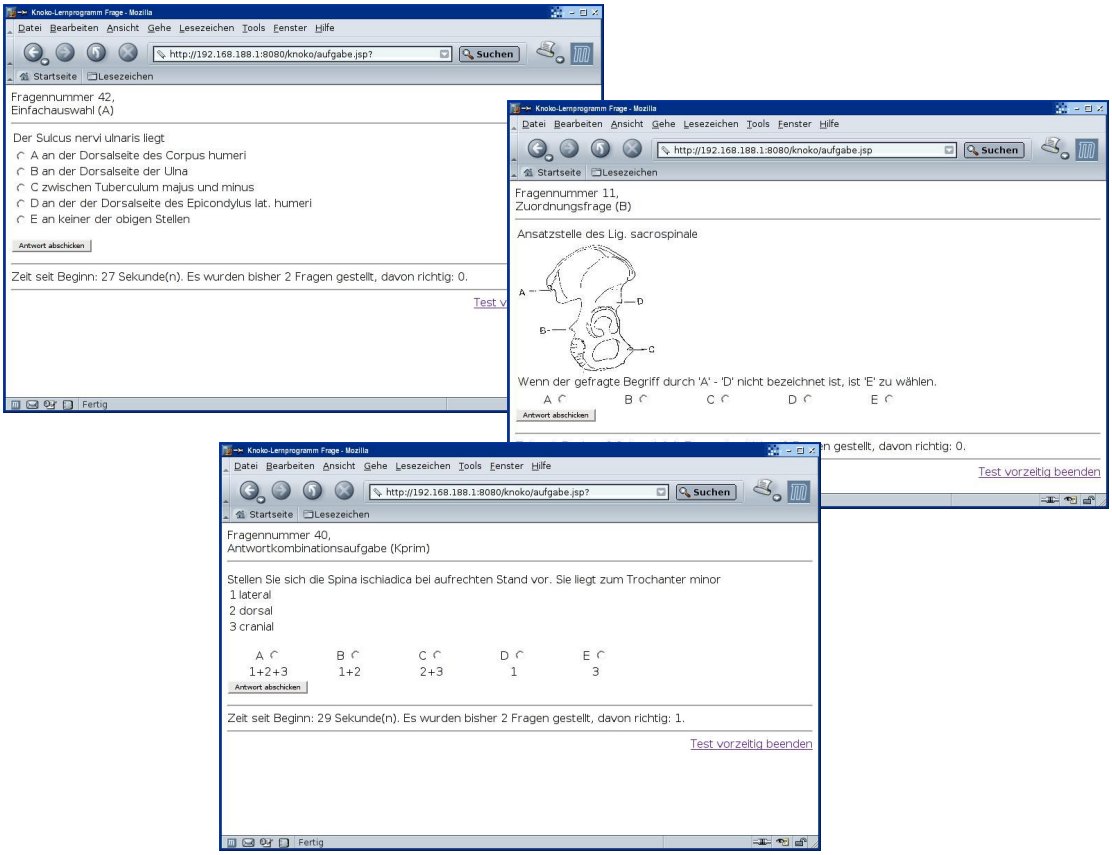

Abbildung 4-3
Screenshots der drei Fragentypen (JSTL/JSP)

Auch bei der Controller-JSP `korrektur.jsp` findet zuerst die Überprüfung statt ob auf Testverlauf zugegriffen werden kann.
Wenn das Objekt Testverlauf vorhanden ist dann seiner Eigenschaft `antwort` der Wert des Parameters `antwort` zugewiesen und anschließend mit einem Funktionsaufruf die Benutzerantwort auf Korrektheit überprüft. Wenn der in der Variablen `korrektur` gespeicherte Rückgabewert "`richtig`" ist wird `richtig.jsp` aufgerufen – andernfalls `falsch.jsp`.

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fw"   uri="http://akh-wien.ac.at/functionwrapper" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<jsp:setProperty name="Testverlauf" property="antwort" />

<c:set var="korrektur" value="${fw:aufgabekorrigieren(Testverlauf)}" />

<c:choose>
  <c:when test="${ korrektur eq 'richtig'}" >
    <jsp:forward page="richtig.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="falsch.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 4-8
Ausschnitt aus `korrektur.jsp`

Die View-JSP `richtig.jsp` gibt „Richtig!“ aus sowie gegebene Antwort, richtige Antwort und zusätzliche Informationen.
Die Checkbox ob die Frage nochmals gestellt werden soll ist nicht aktiviert. Bei der Anforderung für die nächste Frage durch den Benutzer wird der Wert des Parameters aufgabeWiederholen an die Controller-JSP `aufgabe.jsp` übergeben.

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf" class="at.ac.akhwien.mvc.model.Testverlauf"
scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <body>

    Fragennummer  ${Testverlauf.aufgabe.nr}
    <hr />
    <h1>Richtig!</h1>

    <p>Gegebene Antwort: ${Testverlauf.antwort},
       richtige Antwort: ${Testverlauf.aufgabe.loesung} </p>

    <form METHOD="GET" ACTION="aufgabe.jsp">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true" />
        diese Aufgabe nocheinmal stellen
      </p>
```

```
      <input type="submit" value="weiter">
    </form>

    <hr />

    Zeit seit Beginn: ${Testverlauf.dauer} Sekunde(n).
    Es wurden bisher ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    davon richtig: ${Testverlauf.richtigeAufgaben}.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 4-9
Ausschnitt aus `richtig.jsp`

Die Seiten `richtig.jsp` und `falsch.jsp` sind im Grunde ident. Sie unterscheiden sich nur dadurch, dass bei `falsch.jsp` „Falsch!“ ausgegeben wird und im `<input>`-Tag das Attribut `checked` den Wert `"checked"` enthält.

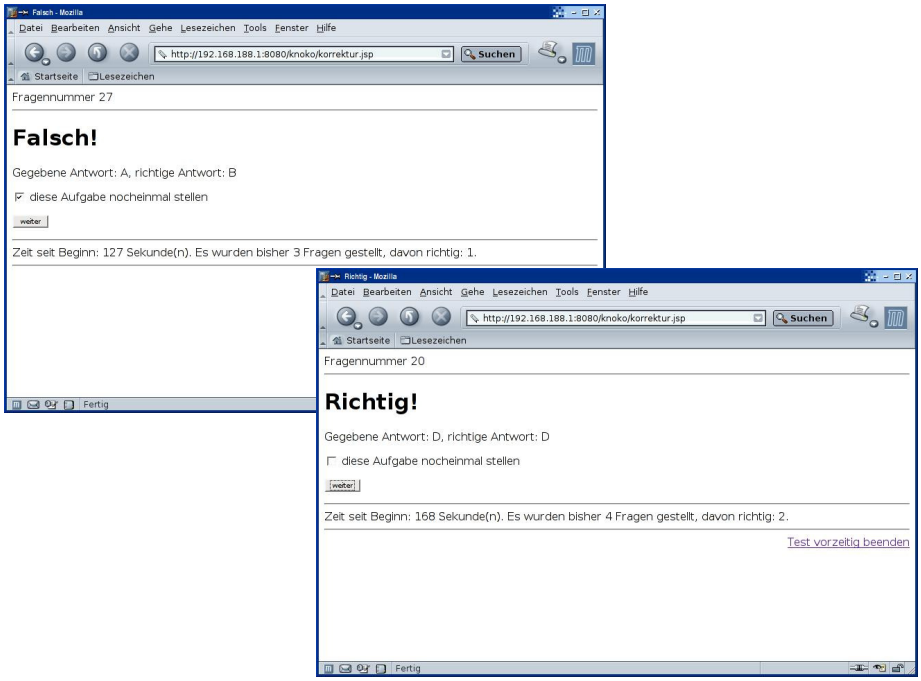

Abbildung 4-4
Screenshots von `richtig.jsp` *und* `falsch.jsp` *(JSTL/JSP)*

Die für die Statistik von `ende.jsp` notwendigen Rechenoperationen lassen sich durch Einsatz der EL leicht durchführen. Mit Hilfe des Tags `<fmt:formatNumber>` der Format-Tag-Library der JSTL können die Vorkommastellen und Nachkommastellen auf je 2 begrenzt werden.
Am Ende der Seite, wenn keine Zugriffe auf Testverlauf mehr folgen, kann die Sitzung für ungültig erklärt werden, damit sind das Session-Objekt und alle in ihm enthaltenen Objekte gelöscht.

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt"  uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>
```

```
<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <body>

    <h1>Ende</h1>
    <h2>Auswertung:</h2>

    <p>
    Vom Computer wurden ${Testverlauf.testumfang} verschieden Fragen
    aus einem Pool von ${Testverlauf.katalog.anzahl} Aufgaben per Zufall
    ausgesucht.
    </p>

    <p>
    Die Dauer zur Beantwortung aller Fragen betrug ${Testverlauf.dauer}
    Sekunde(n).
    Dies entspricht ${Testverlauf.dauer/Testverlauf.gestellteAufgaben} Sekunden
    pro Frage.
    </p>

    <p>
    Insgesammt wurden ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    von denen Sie ${Testverlauf.richtigeAufgaben} richtig beantwortet haben
    ( <fmt:formatNumber value="${ 100*Testverlauf.richtigeAufgaben /
                                  Testverlauf.gestellteAufgaben}"
                        maxIntegerDigits="2" maxFractionDigits="2"/> % )
    </p>


    <hr />
    <a href="index.jsp">zum Start</a>

  </body>
</html>

<sess:invalidate/>
```

Beispiel 4-10
Ausschnitt aus `ende.jsp`

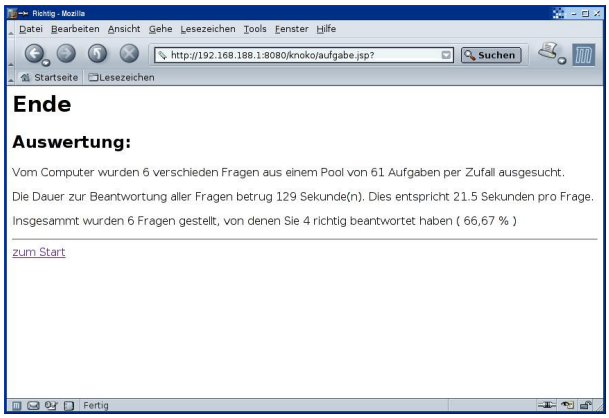

Abbildung 4-5
Screenshot von `ende.jsp` *(JSTL/JSP)*

Controller und Views in der Struts-Version

Die hier vorgestellten Beispiele basieren auf Struts 1.2.8 und setzen einen Servlet Container voraus der die Servlet-API ab Version 2.2 und JSPs ab Version 1.1 unterstützt.
Struts baut auf ANTLR und den Jakarta Commons Projekten Beanutils, Digester, FileUpload, Logging, Validator und Jakarta ORO auf.
Um zeigen zu können was Struts kann und was Struts alleine nicht kann wird in dieser Version gänzlich auf die Möglichkeiten der JSTL und der EL verzichtet. In der Praxis können und sollten sowohl die JSTL und die EL in Struts-Projekten eingesetzt werden.

Installation

Das Java-Archiv `struts.jar` und die Bibliotheksdateien `antlr.jar`, `commons-beanutils.jar`, `commons-digester.jar`, `commons-fileupload.jar`, `commons-logging.jar`, `commons-validator.jar`, `jakarta-oro.jar` müssen in `/WEB-INF/lib/` der Web-Applikation kopiert werden.
Weiters werden in `/WEB-INF/`die Tag-Library-Deskriptoren `struts-bean.tld`, `struts-html.tld`, `struts-logic.tld`, `struts-nested.tld` und `struts-tiles.tld` benötigt.
Die Konfigurationsdateien `struts-config.xml`, `tiles-defs.xml` und `validator-rules.xml` müssen sich ebenfalls in `/WEB-INF/` befinden.
Hilfreich hierbei ist die in der Struts-Binary-Distribution befindliche leere Web-Applikation `struts-blank.war`, da sie alle notwendigen Bibliotheken und Konfigurationsdateien enthält.

Funktionsweise [38] [39] [40]

Das Design des Frameworks Struts war von Anfang an am MVC-Paradigma orientiert. Im Mittelpunkt steht das `ActionServlet`, das als Controller fungiert und bei jedem Request an die Web-Applikation aufgerufen wird. Hierbei übernehmen Request-Processor und die Action-Klassen den Hauptteil der Arbeit zur Abarbeitung des Requests: den Datenaustausch mit dem Model und die Weiterleitung auf die View-Seiten. Darüber hinaus kümmert sich das `ActionServlet` um Initialisierung und Aufräumung von Ressourcen. Das `ActionServlet` wird in `web.xml` mit der Option `<load-on-startup>` deklariert, damit es beim Start der Web-Applikation automatisch geladen, instanziert und die `init()`-Methode aufgerufen wird. Als erstes wird die Konfigurationsdatei (`struts-config.xml` in /WEB-INF/) geladen. Das `ActionServlet` erhält das `ServletMapping *.do`. D.h. bei jedem Request der auf `.do` endet wird die `service()`-Methode des `ActionServlets` gestartet.

```
<servlet>
  <servlet-name>action</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>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>detail</param-name>
    <param-value>2</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
```

```
</servlet>

<!-- Standardmaessiges Action Servlet Mapping mit *.do -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Beispiel 4-11
Servlet-Deklaration in `web.xml`

Weiters werden in `web.xml` die Struts-Tag-Libraries geladen. Da einige der Funktionen, die die Struts-Tag-Library bietet auch mit Hilfe der JSTL realisiert werden können, empfiehlt das Struts Team, wenn möglich, den Standard-Tags gegenüber den Struts-Tags den Vorzug zu geben. Dies betrifft vor allem die Bean- und die Logic-Tag-Library. Aus Gründen der Anschaulichkeit werden in dieser Beispiel-Applikation ausschließlich die Struts-Tags verwendet. Die HTML-Library dient zur Erstellung von Struts Eingabeformularen und somit als Brücke zwischen den `FormBeans` und den JSPs – siehe unten.
Struts-Tiles ist ein Plug-In von Struts, das explizit aktiviert werden muss Es bietet ermöglicht mehrere Inhalte wie Kacheln zu einer einzigen JSP zusammenzufügen (ohne die Verwendung von Frames). Die Information welche Seiten eingefügt werden, ist dabei nicht in der Gesamt-JSP enthalten sondern in eine eigene Konfigurationsdatei ausgelagert (`tiles-defs.xml`). Obwohl das Tiles-Plugin eine gute Unterstützung zur Auslagerung und Trennung von Inhalten in Views darstellen, geht seine Zielsetzung darüber hinaus. Die Funktionalität die das Tiles-Framework bietet sollte daher im Vergleich mit der der Portlets Spezifikation [41] [42] oder Cocoon [43] behandelt werden und wird daher hier nicht weiter besprochen..

```
<taglib>
  <taglib-uri>/tags/struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-html</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-nested</taglib-uri>
  <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-tiles</taglib-uri>
  <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
```

Beispiel 4-12
Taglib-Deklaration `web.xml`

Struts verwendet zur Verarbeitung von Formularen die Struts Form-Beans die von `org.apache.struts.action.ActionForm`.abgeleitet werden. Diese Beans sind mit einer Action verknüpft (siehe unten bei `<action-mappings>` und `<action>`). Wenn die Namen der Eigenschaften mit den beim `Action-Request` übermittelten Parameternamen

übereinstimmen, werden vom `ActionServlet` den Eigenschaften automatisch mit den entsprechenden Werten übergeben bevor die Action geladen und abgearbeitet wird.
Struts bietet mit `ActionForms` noch zusätzliche Möglichkeiten. `ActionForms` können vor Abarbeiten der Action eine Validierungs-Funktion automatisch aufrufen, die die Eigenschaften auf Korrektheit, gemäß definierter Vorgaben, überprüft. Im Fehlerfall wird automatisch ein Forward zurück auf die JSP mit dem Eingabeformular vorgenommen. Es können auch die passenden Fehlermeldungen ausgegeben werden, sofern diese definiert wurden.

```
<form-beans>
  <form-bean      name="AntwortForm"
                  type="at.ac.akhwien.mvc.formbeans.AntwortFormBean"/>
  <form-bean      name="WiederholenForm"
                  type="at.ac.akhwien.mvc.formbeans.WiederholenFormBean"/>
</form-beans>
```

Beispiel 4-13
Form-Beans-Deklaration in `struts-config.xml`

Man kann Forwards definieren die von jeder Action aus aufgerufen werden können.

```
<global-forwards>
  <forward name="index" path="/index.jsp" />
</global-forwards>
```

Beispiel 4-14
Global-Forwards-Definition in `struts-config.xml`

Entsprechend obigem `<servlet-mapping>` wird jedes Mal, wenn von einem Client ein URI mit der Endung `".do"` aufgerufen wird, das ActionServlet zur Bearbeitung der Anfrage gestartet. Das ActionServlet wiederum vergleicht dann, ob der URI vor dem `".do"` mit einem `<action>`-Eintrag im Abschnitt `<action-mappings>` der Konfigurationsdatei übereinstimmt. Ist ein passender Eintrag vorhanden, lädt das `ActionServlet` die entsprechende „`Action-Klasse`“ (= „`Action`“) und ruft die `execute()`–Methode auf. Diese `execute()` Methode wird vom Programmierer implementiert – siehe unten. Die Action wird über `name` mit einer `Form-Bean` verknüpft und es werden die Views definiert zu denen weitergeleitet werden soll. Wenn sich die Namen der Views ändern sollten, muss das `Action-Servlet` nicht geändert und neu compiliert werden.
Weiters kann hier definiert werden ob die `Form-Bean` automatisch vom `Validator` geprüft werden soll.
.

```
<action-mappings>

  <action    path="/aufgabe"
             type="at.ac.akhwien.mvc.actions.Aufgabe"
             scope="request"
             name="WiederholenForm" >
    <forward name="neu"  path="/aufgabenstellung.jsp"/>
    <forward name="ende"  path="/ende.jsp"/>
  </action>

  <action    path="/korrektur"
             type="at.ac.akhwien.mvc.actions.Korrektur"
             scope="request"
             name="AntwortForm" >
    <forward name="richtig"  path="/richtig.jsp"/>
    <forward name="falsch"   path="/falsch.jsp"/>
```

```
    </action>


  </action-mappings>
```

Beispiel 4-15
Actionmappings-Defnition in `struts-config.xml`

Um die Ausgabe von Text in Web-Applikationen auf einfache Weise zu internationalisieren bietet Struts die Möglichkeit Zeichenketten in Ressourcen-Dateien auszulagern, wobei für jedes Land bzw. jede Landessprache eine eigene Datei angelegt werden kann. Die Unterscheidung verschiedener Sprachen geschieht entsprechend der vom Browser gelieferten Landes/Sprach-Codes (English = `en`, Deutsch, allgemein = `de`, Österreich = `de-AT`, usw.). Die Dateien müssen sich im Klassen-Pfad der Web-Applikation befinden. Beim Schreiben dieser Web-Applikation stellte sich heraus, dass die Datei `resources.ApplicationResources` in `web.xml` definiert und im Verzeichnis `at/ac/akh-wien/mvc` vorhanden sein muss, auch wenn sie, wie in diesem Beispiel, leer ist, da das `ActionServlet` sonst eine Fehlermeldung erzeugt.

```
  <message-resources parameter="resources.ApplicationResources" />
```

Beispiel 4-16
Actionmappings-Defnition in `struts-config.xml`

Die Homepage des Lernprogramms nimmt nicht an Sessions teil und verweist auf den Start des Lernprogramms, die Action `aufgabe`.

```
<%@page  session="false" %>

<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabe.do">Start</a>.

    </hr>

  </body>
</html>
```

Beispiel 4-17
`index.html`

Wird die Action `aufgabe` von einer der Seiten `richtig.jsp` oder `falsch.jsp` aus aufgerufen, übergeben diese den Parameter ob eine Aufgabe noch einmal gestellt werden soll. Falls also der Parameter "`aufgabeWiederholen`" übergeben wurde, erzeugt das `ActionServlet` ein `FormBean`-Objekt und weist den Wert der Eigenschaft `aufgabeWiederholen` zu.

```
package at.ac.akhwien.mvc.formbeans;

import org.apache.struts.action.ActionForm;

public final class WiederholenFormBean extends ActionForm
{
  // Eigenschaften
  private String aufgabeWiederholen = "";

  public void setAufgabeWiederholen (String aufgabeWiederholen_neu)
  {
    this.aufgabeWiederholen = aufgabeWiederholen_neu;
  }

  public String getAufgabeWiederholen ()
  {
    return this.aufgabeWiederholen;
  }
}
```

Beispiel 4-18
Ausschnitt aus `WiederholenFormBean.jsp`

Die Methode `execute()` der Action `aufgabe` bekommt im Vergleich zu einem Servlet zusätzlich zu `HttpRequest` und `HttpResponse` Parameter vom Typ `ActionMapping` und `ActionForm` übermittelt. Der Parameter `mapping` und `form` enthalten die in `struts-config.xml` definierten Informationen.

Das `ActionServlet` erledigt bei einer Anfrage automatisch der Reihe nach oft benötigten Aufgaben, die sonst wiederholt implementiert werden müssten. Dies sind unter anderen

- Setzen von ContentType
- Setzen von Ländercodes (entsprechend der vom Browser gesendeten `accept-Language` und der vorhandenen `Resource-Bundles`)
- Befüllen der Beans-Eigenschaften mit Parameterwerten
- Validierung
- Ausführen von Actions
- Weiterleiten

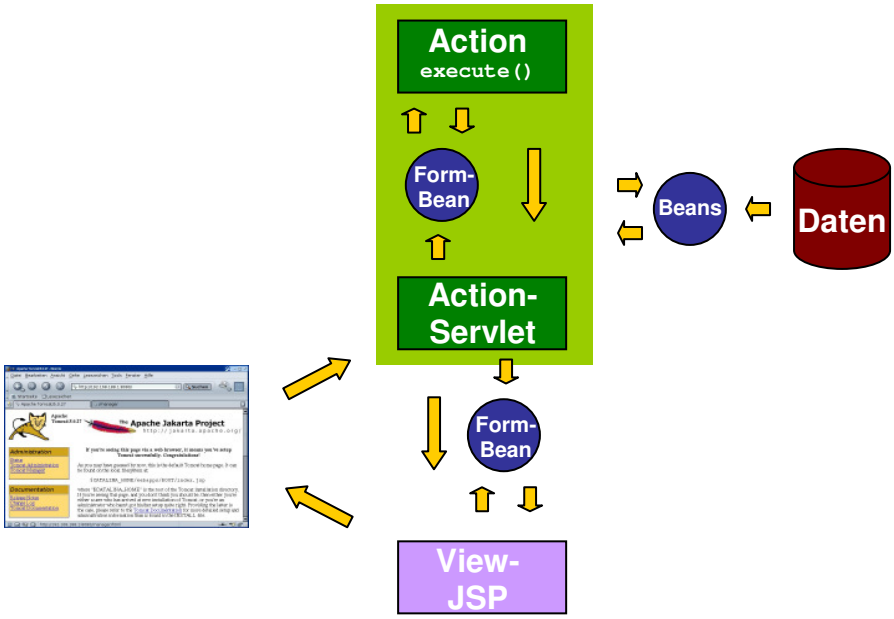

Abbildung 4-6
MVC-Beispiel mit Stuts

In der kommenden überarbeiteten Struts-Version 1.3 wurde diese monolithische durch eine modulare Architektur entsprechend dem *Chain of Responsibility Pattern* (CoR) ersetzt. Teilaufgaben können entsprechend den Angaben in einer Konfigurationsdatei eingefügt, entfernt oder durch neue, ersetzt werden.

Hier teilt die Action zu Beginn mit, dass sie an der Session teilnehmen will. Danach wir überprüft ob das `Aufgabenkatalog`-Objekt im `ServletContext` (=Scope `application`) vorhanden ist. Wenn nicht, wird es erstellt und danach sofort die Methode `setDateiname()` mit dem Dateinamen aufgerufen, den der Listener `SCInit` im `ServletContext` unter "`dateiname`" abgelegt hat.
Weiters wird versucht auf das `session`-Objekt `Testverlauf` zuzugreifen. Falls noch nicht vorhanden, wird eines im Scope `session` angelegt und sofort danach die Methode `setKatalog()` mit einer Referenz auf das Objekt `Aufgabenkatalog` und die Methode `setTestumfang` mit dem in `web.xml` definierten Initialisierungswert `testumfang` aufgerufen.

Falls der Parameter "`aufgabeWiederholen`" dem `ActionForm` übergeben wurde, wird sein Wert der Variablen `wiederholen` zugewiesen und, wenn "`true`", veranlasst, dass die Aufgabe wieder gestellt wird.
Danach wird eine neue Aufgabe angefordert. Der Rückgabewert wird in der Variablen `weiter` gespeichert und anschließend der in `struts-config.xml` definierte Forward veranlasst (`"neu" -> aufgabenstellung.jsp oder "ende" -> ende.jsp`).

```
public final class Aufgabe extends Action
{
  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) throws Exception
  {
```

```
    Aufgabenkatalog                   aufgabenkatalog;
    Testverlauf                       testverlauf;
    at.ac.akhwien.mvc.model.Aufgabe   aufgabe;
    ServletConfig                     config;
    ServletContext                    application;
    HttpSession                       session;
    String              wiederholen;
    String              weiter;


    config = getServlet().getServletConfig();
    application = config.getServletContext();
    session=request.getSession(false);

    aufgabenkatalog = (at.ac.akhwien.mvc.model.Aufgabenkatalog)
                      application.getAttribute("Aufgabenkatalog");
    if (aufgabenkatalog==null)
    {
      aufgabenkatalog = new Aufgabenkatalog();
      aufgabenkatalog.setDateiname( (java.lang.String)
                                    application.getAttribute("dateiname") );
      application.setAttribute("Aufgabenkatalog",aufgabenkatalog);
    }

    testverlauf= (at.ac.akhwien.mvc.model.Testverlauf)
                 session.getAttribute("Testverlauf");
    if (testverlauf==null)
    {
      testverlauf = new Testverlauf();
      testverlauf.setKatalog((at.ac.akhwien.mvc.model.Aufgabenkatalog)
                             aufgabenkatalog.getReferenz());
      testverlauf.setTestumfang(Integer.parseInt(
                                application.getInitParameter("testumfang")));
      session.setAttribute("Testverlauf",testverlauf);

    }
    else
    {
      wiederholen = (String) PropertyUtils.getSimpleProperty
                                                  (form, "aufgabeWiederholen");
      if (wiederholen.equalsIgnoreCase("true"))
      {
        testverlauf.setAufgabeWiederholen(true);
      }
    }

    weiter=testverlauf.neueAufgabe();

    return (mapping.findForward(weiter));


  }

}
```

Beispiel 4-19
Ausschnitt aus der Action `Aufgabe.java`

Die JSP `aufgabenstellung.jsp` ist hier wie auch in der JSTL-Version für das Layout der verschieden Fragentypen zuständig.
Zur Ausgabe der Eigenschaften der aktuellen `Aufgabe`-Bean werden Struts-Bean-Tags verwendet:
z.B. entspricht `<bean:write name="Testverlauf" property="aufgabe.nr"/>` dem EL-Ausdruck `${Testverlauf.aufgabe.nr}` bzw. dem Scriptlet
`<%= Testverlauf.getAufgabe().getNr() %>`.

Diese Tag-Libray kann jedoch nicht überall eingesetzt werden. Beim Aufruf der Bilddatei muss wieder auf Scriptlet-Code zurückgegriffen werden.
Der `<form>-Tag` von HTML wird durch den `<html:form>`-Tag ersetzt. Es erfolgt durch die Verwendung der Struts-Tags implizit auch eine Anmeldung der in `struts-config.xml` angegebenen `Action-Form`-Bean.
Der `<html:text name="`*Form-Bean*`" property="`*Eigenschaft*`">`-Tag (das Äquivalent zum HTML-Tag `<input type="text" >`) würde dann z.B. automatisch in das `value`-Attribut den Wert der entsprechendem Eigenschaft der `FormBean` eintragen. Dies ist hilfreich wenn bei Fehleingaben wieder auf die Eingabeseite zurückgeleitet wird, da der Benutzer nicht mehr alles erneut eintippen muss. Diese Funktion ist hier nicht notwendig und auch nicht gewünscht. Es werden daher der original HTML-Tag `<input type="radio">` verwendet.
Die Action `korrektur` wird mit dem Parameter `antwort` aufgerufen wenn der Benutzer den `Submit`-Button anklickt.

Entsprechend dem Fragetyp (`modus`) wird das entsprechende Layout ausgegeben. Die Logic-Tag-Library bietet ausschließlich Tags die nur eine Bedingung prüfen können – ähnlich dem if-Tag der JSTL. Daher ist für jeden möglichen Wert von `modus` eine eigene Überprüfung mit `<logic:equal>` notwendig.
Ist der Bilddateiname nicht leer `<logic:notequal … value="">`, wird die Bilddatei ausgegeben (betrifft nur Fragetypen A und B).

```
<%@page  session="true" %>

<%@ taglib prefix="html"  uri="/tags/struts-html" %>
<%@ taglib prefix="bean"  uri="/tags/struts-bean" %>
<%@ taglib prefix="logic" uri="/tags/struts-logic" %>

<html>
  <body bgcolor=white>
    Fragennummer  <bean:write name="Testverlauf" property="aufgabe.nr"/>
      <html:form action="/korrektur.do">
        <logic:equal name="Testverlauf" property="aufgabe.modus" value="1">
          Einfachauswahl (A)
          <hr/>

          <table>
            <tr>
              <td style="zeile">
                <bean:write name="Testverlauf"
                            property="aufgabe.fragenintro" filter="false" />
              </td>
            </tr>
            <logic:notEqual name="Testverlauf"
                            property="aufgabe.bildname" value="">
                <tr>
                  <td >
                    <img src="<%= Testverlauf.getAufgabe().getBildname()%>" />
                  </td>
                </tr>
            </logic:notEqual>
          </table>

          <table>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="A"> </td>
                 <td> A </td>
                 <td>
                   <bean:write name="Testverlauf"
                               property="aufgabe.wahlantwort1"/>
                 </td>
```

```
            </label>
          </tr>
          <tr>
            <label>
               <td> <input type="radio" name="antwort" value="B"> </td>
               <td> B </td>
               <td>
                 <bean:write name="Testverlauf"
                             property="aufgabe.wahlantwort2"/>
               </td>
            </label>
          </tr>
          <tr>
            <label>
               <td> <input type="radio" name="antwort" value="C"> </td>
               <td> C </td>
               <td>
                 <bean:write name="Testverlauf"
                             property="aufgabe.wahlantwort3"/>
               </td>
            </label>
          </tr>
          <tr>
            <label>
               <td> <input type="radio" name="antwort" value="D"> </td>
               <td> D </td>
               <td>
                 <bean:write name="Testverlauf"
                             property="aufgabe.wahlantwort4"/>
               </td>
            </label>
          </tr>
          <tr>
            <label>
               <td> <input type="radio" name="antwort" value="E"> </td>
               <td> E </td>
               <td>
                 <bean:write name="Testverlauf"
                             property="aufgabe.wahlantwort5"/>
               </td>
            </label>
          </tr>
        </table><p></p>
      </logic:equal>

      <logic:equal name="Testverlauf" property="aufgabe.modus" value="2">
        Antwortkombinationsaufgabe (Kprim)

        ...

      </logic:equal>

      <logic:equal name="Testverlauf" property="aufgabe.modus" value="3">
        Zuordnungsfrage (B)

        ...

      </logic:equal>

      <html:submit property="submit" value="Antwort abschicken"/>
    </html:form>

  </hr>

  Zeit seit Beginn: <bean:write name="Testverlauf" property="dauer"/>
  Sekunde(n).
  Es wurden bisher <bean:write name="Testverlauf"
                               property="gestellteAufgaben"/> Fragen gestellt,
  davon richtig: <bean:write name="Testverlauf" property="richtigeAufgaben"/>.
```

```
    <hr />

    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 4-20
`aufgabenstellung.jsp`, *gekürzt*

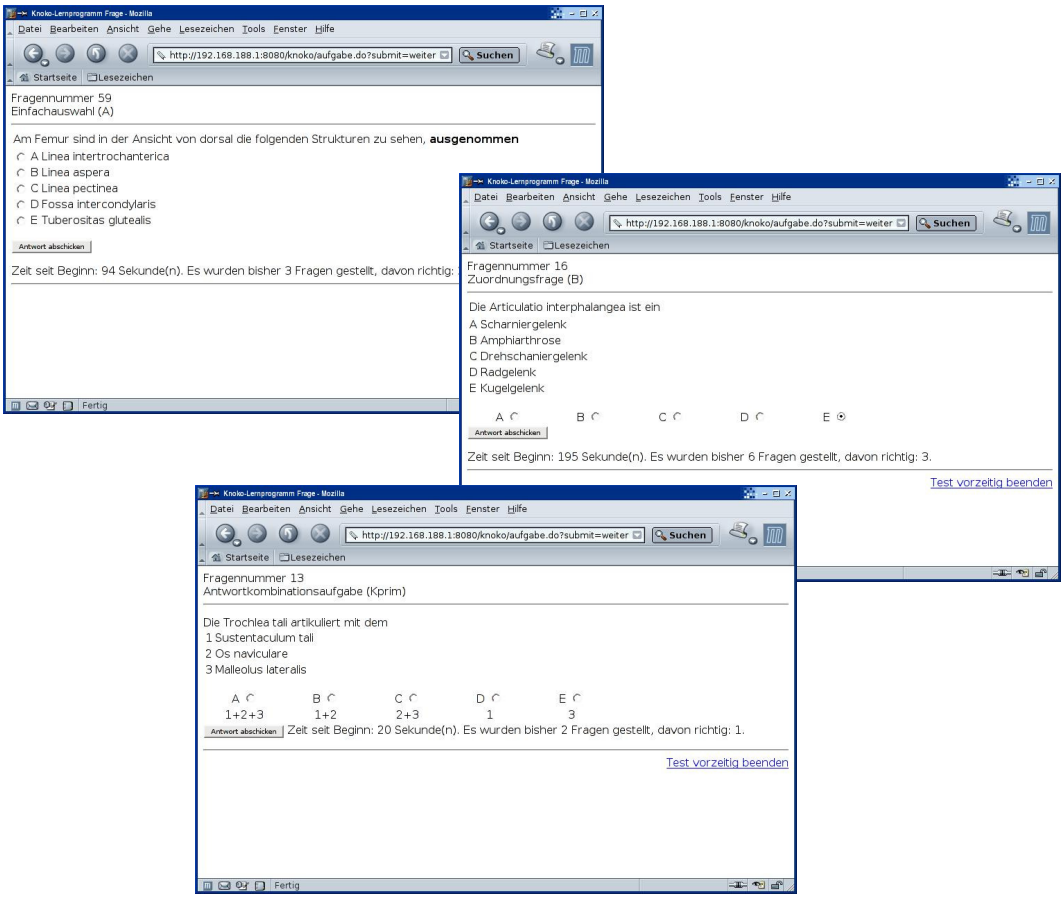

Abbildung 4-7
Screenshots der drei Fragentypen (Struts)

Die `ActionServlet` erhält mit dem Parameter antwort die Benutzerantwort, erzeugt ein `FormBean`-Objekt und weist den erhaltenen Wert der Eigenschaft `antwort` zu.

```
package at.ac.akhwien.mvc.formbeans;

import org.apache.struts.action.ActionForm;

public final class AntwortFormBean extends ActionForm
{
  // Eigenschaften
  private String antwort = "";

  // Methoden

  public void setAntwort (String antwort_neu)
  {
    this.antwort = antwort_neu;
  }
```

```
  public String getAntwort ()
  {
    return this.antwort;
  }
}
```

Beispiel 4-21
Ausschnitt aus `AntwortFormBean.jsp`

Nachdem das `ActionServlet` die Form-Bean erzeugt und befüllt hat, wird die `exectue()` Methode der Action `korrektur` aufgerufen. Wenn das `Testverlauf`-Objekt aus dem `Session`-Objekt gelesen werden kann, wird der Wert der Eigenschaft antwort der Form-Bean gelesen, dem `Testverlauf`-Objekt übergeben und dieses aufgefordert die Antwort auf Korrektheit zu prüfen. Das Ergebnis wird verwendet um auf die in `struts-config.xml` definierte View-Seite weiterzuleiten.

```
package at.ac.akhwien.mvc.actions;

import java.io.*;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.commons.beanutils.PropertyUtils;
import at.ac.akhwien.mvc.model.Aufgabenkatalog;
import at.ac.akhwien.mvc.model.Testverlauf;

public final class Korrektur extends Action
{
  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) throws Exception
  {
    Testverlauf         testverlauf;
    HttpSession         session;
    String              antwort;
    String              weiter;

    session=request.getSession(false);

    testverlauf= (at.ac.akhwien.mvc.model.Testverlauf)
                                          session.getAttribute("Testverlauf");
    if (testverlauf==null)
    {
      return(mapping.findForward("index"));
    }

    antwort = (String) PropertyUtils.getSimpleProperty(form, "antwort");

    testverlauf.setAntwort(antwort);

    weiter=testverlauf.aufgabekorrigieren();

    return (mapping.findForward(weiter));
  }
}
```

Beispiel 4-22
Ausschnitt aus der Action `korrektur.java`

Die View-JSP `richtig.jsp` entspricht wieder weitgehend der JSTL-Version und gibt „Richtig!“, die gegebene Antwort, die richtige Antwort und zusätzliche Informationen aus. Auch hier werden Bean-Tag-Library Tags dazu verwendet. Der `<form>` Tag wurde gegen einen `<html:form>`-Tag getauscht.
Die Checkbox ob die Frage nochmals gestellt werden soll ist nicht aktiviert. Bei der Anforderung für die nächsten Frage durch den Benutzer wird der Wert des Parameters `aufgabeWiederholen` an die Action `aufgabe` übergeben.

```
<%@page  session="true" %>


<%@ taglib prefix="html"  uri="/tags/struts-html" %>
<%@ taglib prefix="bean"  uri="/tags/struts-bean" %>

<html>
  <body bgcolor=white>

    Fragennummer: <bean:write name="Testverlauf" property="aufgabe.nr"/>
    <hr />
    <h1>Richtig!</h1>

    <p>Gegebene Antwort: <bean:write name="Testverlauf" property="antwort"/>,
       richtige Antwort: <bean:write name="Testverlauf"
                                     property="aufgabe.loesung"/> </p>

    <html:form method="GET" action="/aufgabe.do">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true" />
        diese Aufgabe nocheinmal stellen
      </p>
      <html:submit property="submit" value="weiter" />
    </html:form>

    <hr />

    Zeit seit Beginn: <bean:write name="Testverlauf" property="dauer"/>
    Sekunde(n).
    Es wurden bisher <bean:write name="Testverlauf"
                                 property="gestellteAufgaben"/> Fragen gestellt,
    davon richtig: <bean:write name="Testverlauf" property="richtigeAufgaben"/>.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>
    </hr>

  </body>
</html>
```

Beispiel 4-23
Ausschnitt aus `richtig.jsp`

Die Seiten `richtig.jsp` und `falsch.jsp` sind auch hier im Grunde ident. Der Unterschied ist nur, dass bei `falsch.jsp` „Falsch!“ ausgegeben wird und im `<input>`-Tag das Attribut `checked` den Wert `"checked"` enthält.

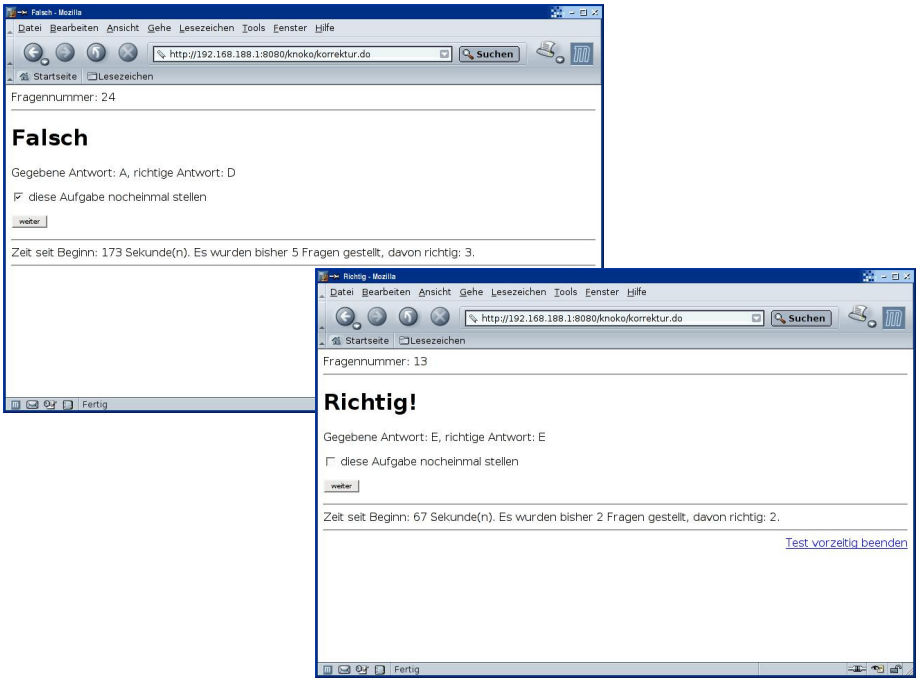

Abbildung 4-8
Screenshots von `richtig.jsp` *und* `falsch.jsp` *(Struts)*

Die notwendigen Rechenoperationen für die Statistik in ende.jsp sind hier nur mit der Hilfe von Scriptlets zu realisieren, da mit Bean-Tags alleine dies nicht möglich ist. Hier sollte wieder auf die Funktionen der EL zurückgegriffen werden.
Am Ende der Seite, wenn keine Zugriffe auf Testverlauf mehr folgen, kann die Sitzung für ungültig erklärt werden, damit ist das session-Objekt und alle in ihm enthaltenen Objekte gelöscht. Auch dafür wird hier ein Scriptlet verwendet.

```
<%@page  session="true" %>

<%@taglib prefix="bean"  uri="/tags/struts-bean" %>

<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <% session.invalidate(); %>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <head>
    <title></title>
  </head>
  <body bgcolor=white>
    <h1>Ende</h1>
    <h2>Auswertung:</h2>

    <p>
     Vom Computer wurden <bean:write name="Testverlauf" property="testumfang"/>
     verschieden Fragen aus einem Pool von
     <bean:write name="Testverlauf" property="katalog.anzahl"/>
     Aufgaben per Zufall ausgesucht.
    </p>

    <p>
```

```
     Die Dauer zur Beantwortung aller Fragen betrug
     <bean:write name="Testverlauf" property="dauer"/> Sekunde(n). Dies
     entspricht <%= Testverlauf.getDauer()/Testverlauf.getGestellteAufgaben() %>
     Sekunden pro Frage.
    </p>

    <p>
     Insgesammt wurden
     <bean:write name="Testverlauf" property="gestellteAufgaben"/> Fragen
     gestellt, von denen Sie
     <bean:write name="Testverlauf" property="richtigeAufgaben"/>  richtig
     beantwortet haben
     ( <%= 100*Testverlauf.getRichtigeAufgaben() /
                                        Testverlauf.getGestellteAufgaben() %> % )
    </p>


    <hr />
    <a href="index.jsp">zum Start</a>

  </body>
</html>

<% session.invalidate(); %>
```

Beispiel 4-24
Ausschnitt aus `ende.jsp`

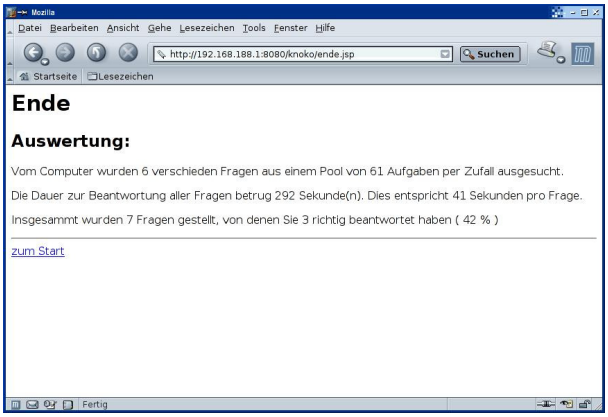

Abbildung 4-9
Screenshot von `ende.jsp` *(Struts)*

Controller und Views in der JSF-Version

Die JavaServer Faces-Version setzt Servlet-API Version 2.3 oder höher und JSP Version 1.2 oder höher voraus. Sun empfiehlt den Einsatz eines JSP 2.0 kompatiblen Containers.
Neben der Referenz-Implementation von Sun Microsystems kann alternativ auch auf das MyFaces-Projekt der Apache Software Foundation verwendet werden. MyFaces ist eine OpenSource-Implementation der JSF-Spezifikation. Beide bauen auf weiteren Apache-Commons-Bibliotheken auf.

Installation

Folgende Java-Archive müssen in `/WEB-INF/lib/` der Web-Applikation vorhanden sein: die JSF-Bibliotheken von Sun-Microsystems Version 1.0 oder größer `jsf-api.jar` und `jsf-impl.jar`. Die Dateien sind in der einzeln verfügbaren JSF-Referenz-Implementation (JSF RI) oder im sehr viel umfangreicheren Web-Services-Development-Pack (WSDP) enthalten.
Die JSF-Dateien bauen auf den Jakarta-Commons-Bibliotheksdateien `commons-beanutils.jar`, `commons-collections.jar`, `commons-digester.jar`, `commons-logging.jar` auf. Weiters werden empfohlen die JSTL-Bibliotheken `jstl.jar` und `standard.jar`, in der JSP 2.0-kompatiblen Version 1.1.
Die JSF Referenzimplementation enthält einige Beispiel-Web-Applikationen, die alle notwendigen Bibliotheken und Konfigurationsdateien enthält.

Für MyFaces benötigt man die Dateien `myfaces-api.jar` und `myfaces-impl.jar`, die auf folgenden zusätzlich Bibliotheken aufbauen: `commons-beanutils.jar`, `commons-codec.jar`, `commons-collections.jar`, `commons-digester.jar`, `commons-el.jar`, `commons-validator.jar`, `jakarta-oro.jar`.
MyFaces bietet zusätzliche Komponenten in Form der optionalen Tomahawk-Bibliothek (`tomahawk.jar`), die hier nicht benötigt wird.
Auch bei MyFaces wird der Einsatz von JSTL Version 1.1 empfohlen.
MyFaces enthält eine leere Beispiel-Web-Applikation (`myf-blank.war`), die alle notwendigen Bibliotheken und Konfigurationsdateien bereitstellt.

Bei diesem MVC-Beispiel benötigt man darüber hinaus die Apache-Taglibs-Bibliothek `taglibs-session.jar`.

Beide JSF-Implementationen benötigen die Konfigurationsdatei `faces-config.xml` im Verzeichnis `/WEB-INF/`.

Funktionsweise [44] [45] [46] [47] [48]

Die Architektur der JavaServer Faces orientiert sich an der moderner grafischer Benutzeroberflächen (Graphical-User-Interface = GUI), die auf Komponenten (wie Texteingabefeld, Button, Optionsliste, Label, usw.) basiert und eine Ereignisverarbeitung (Event-Handler / Event-Listener) besitzen. Tastatureingaben und Maus-Klicks erzeugen hierbei Ereignisse (Events), die von den betroffenen Komponenten verarbeitet werden. Die Programmlogik ist in JavaBeans gespeichert, für die Darstellung der Komponenten werden JSPs mit eigenen Komponenten-Tag-Libraries verwendet, wobei die Tag-Libraries auch als Bindeglied zwischen den Komponenten und den Beans dienen.

Komponenten und Ereignisverarbeitung

Als Beispiel für Ereignisverarbeitung soll eine Verzeichnisstruktur dienen, die in Form eines Baumes organisiert ist - z.B. eine Verzeichnisstruktur auf der Festplatte wie man es von Dateimanagern kennt. Die Komponente, die diesen Verzeichnisbaum im Fenster darstellt, ist eine Tree-View-Komponente. Durch Doppelklick auf die einen Knotenpunkt werden die Zweige des Knotens (die Unterverzeichnisse) und z.B. in einer List-Komponente des Fensters der Inhalt des Verzeichnisses dargestellt. Durch einen einfachen Klick wird nur der Inhalt des gewählten Verzeichnisses dargestellt.

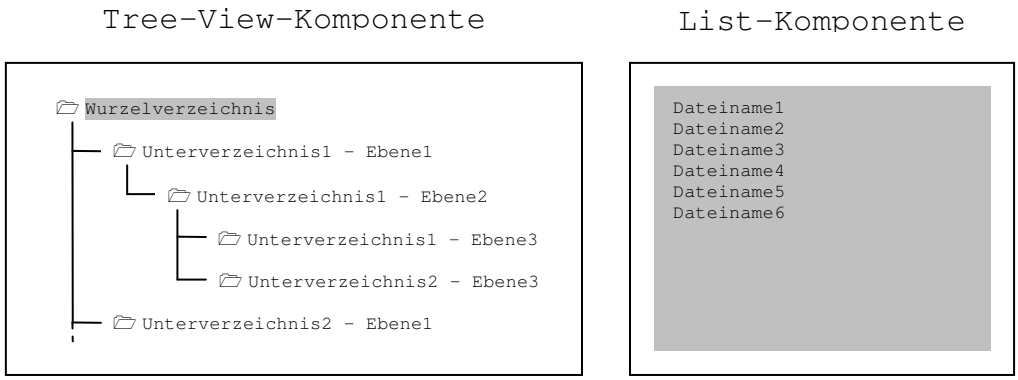

Abbildung 4-10
Beispiel-Komponenten für Ereignisverarbeitung

Aus Sicht des Programms handelt es sich bei den Klicks um Ereignisse (Events) auf die es reagieren muss (Darstellung der Unterverzeichnisse bzw. Darstellung des Inhaltes). Hier handelt es sich um zwei verschiedene Arten von Ereignissen (Doppel- und Einfachklick). Dementsprechend benötigt das Programm zwei verschiedene Arten von Events verarbeiten. Gibt es keine Bearbeitung von Doppelklicks reagiert das Programm auch nicht auf Doppelklicks. Dem Eventhandler wird ein Event-Objekt übergeben. So erfährt er z.B. was im Baum angeklickt wurde. Beim Doppelklick wird die Tree-View-Komponente neu gezeichnet. Beim Einfachklick wird ein Event an die List-Komponente geschickt, die ihrerseits wieder einen Eventhandler besitzt, der dafür sorgt, dass die Listenkomponente den Verzeichnisinhalt darstellt.

Die „Ereignisse“, auf die ein Web-Server reagieren soll, können bei einer Web-Applikation im klassischen Fall nur durch einen Request über das Netz an den Event-Handler im servereseitigen Programm übermittelt werden. Wenn man von der Verwendung von client-seitigem JavaScript in Web-Seiten absieht, sind die Events eines Web-Interface das Klicken auf einen Link oder das Betätigen des Submit-Buttons eines Formulars. Das Programm am Web-Server antwortet seinerseits mit einer Response über das Netz, und die Seite wird zur Gänze neu aufbaut.
Wenn also obiges Beispiel in einer Web-Applikation realisiert werden soll, dann könnten die Baumknoten als Links realisiert werden und bei einem Klick wird dem Server übermittelt welcher Link angeklickt wurde, der Server kümmert sich um die aktualisierte Darstellung der Tree-Komponente und der List-Komponente mittels HTML/CSS und übermittelt das Ergebnis an den Client.

Eine andere in letzter Zeit sehr populär gewordene Technik Web-Seiten interaktiver zu gestalten ist „*asynchronous JavaScript and XML*“ (Ajax) [49] [50] [51]. Ajax bedient sich JavaScript und CSS um am Client die Darstellung im Browser anzupassen. Die Ereignisverarbeitung geschieht hierbei am Client. Dieser sendet einen XMLHttpRequest an den Webserver. Ein am Server befindliches Programm bearbeitet die Anfrage und sendet die Antwort in Form von einer XML-Datei zurück an den Client. Die XML-Daten werden vom Browser als DOM (Document Object Model) repräsentiert. Das JavaScript-Programm kann auf diese Daten zugreifen und das UI entsprechend anpassen.
Bei einer Ajax-Version des Baum-Beispiels würde ein JavaScript-Programm, je nachdem welcher Baum-Knoten angeklickt wurde, eine gezielte Anfrage an den Server Schicken. Der Server schickt die Daten über Unterverzeichnisse und Inhalt zurück an den Client.

Clientseitige Java-Script-Programme zeichnen mittels HTML/CSS sowohl den Baum als auch die Liste neu.

JavaServer Faces wurde erdacht um eine komponenten-basierende Ereignisbehandlung nach dem klassischen Fall zu ermöglichen. Allerdings schließt das den Einsatz von Ajax nicht aus [52]. Im Rahmen des MyFaces Projects wurden bereits UI-Komponenten entwickelt, die mit Hilfe von Ajax funktionieren (z.B. Tree oder Suggest).

Die Verarbeitung von Ereignissen und Komponenten-Verwaltung erledigt bei JavaServer Faces das FacesServlet. Wie auch bei Struts wird das FacesServlet bei Start der Web-Applikation automatisch vom Server geladen und die init()-Methode aufgerufen. Auch hier wird das Servlet jedes Mal gestartet wenn eine Anfrage an die Web-Applikation mit /faces/ beginnt.

Das FacesServlet muss den aktuellen Zustand der auf einer Web-Seite verwendeten Komponenten kennen, damit es weiß wie es auf einen Event reagieren soll. Der StateManger kümmert sich um das Speichern und Wiederherstellen der Komponenten. Wie schon beim Sitzungsmanagement erwähnt, handelt es sich bei HTTP um ein zustandsloses Protokoll JSF bietet zwei verschiedene Möglichkeiten den Zustand zu speichern: entweder am Client oder am Server. Die gewünschte Methode kann in web.xml mit dem Context-Parameter `javax.faces.STATE_SAVING_METHOD` definiert werden. Ist der Wert "`server`", wird der Zustand typischerweise in einer HttpSession gespeichert werden. Ist der Wert "`client`" wird der Wert typischerweise in einem *hidden field* gespeichert und mit dem nächsten Formular-Submit an den Server geschickt. Wird nichts in web.xml angegeben ist der Default-Wert "`server`". Beide Methoden bieten die schon beim Sitzungsmanagement behandelten Vor- und Nachteile.

JSF beschränkt den Programmierer auf die vorhandenen Komponenten. JSF ist von seinem Design her dahin konzipiert, dass das Framework um selbst geschriebene Komponenten erweitert werden kann.
Es ist weiters möglich, dass nur der Renderer einer Komponente neu entwickelt wird. So ist es möglich an bereits vorhandene Komponenten weitere Renderer anzuhängen, ohne die Komponentenfunktionen neu implementieren zu müssen.

JSF-Lifecycle

Wird auf einen Link geklickt, der auf eine Seite innerhalb der JSF-Applikation zeigt, handelt es sich um einen "Faces-Request". Zeigt er auf eine Seite, die keine JSF-Seite der Applikation ist (z.B. eine "normale" JSP oder HTML-Seite) ist dies ein Non-Faces-Request.

Wenn der Server eine JSF-Seite liefert handelt es sich um einen Faces-Response. Wird eine Seite geliefert die keine JSF-Seite ist, entspricht dies einem Non-Faces-Response.
Dementsprechend sind aus Sicht der JSF-Applikation folgende 3 Kombinationen möglich

- Non-Faces-Request erzeugt Faces-Response
- Faces-Request erzeugt Faces Response
- Faces-Request erzeugt Non-Faces-Response

Wird eine JSF-Seite aufgerufen wird zuerst der Zustand der View-Komponenten wieder hergestellt oder neu erzeugt (Restore View). Nachdem das FacesServlet den Zustand der Komponenten nun kennt, werden die Request-Parameter verarbeitet und die *Zustände der betroffenen Komponenten* dementsprechend aktualisiert (Apply Request Values). Sollte bei

der Aktualisierung ein Fehler auftreten, wird die Verarbeitung abgebrochen und eine Darstellung der Komponenten (evtl. inkl. Fehlermeldung) erzeugt und an den Client geschickt. JSF bietet wie auch Struts die Möglichkeit eine automatischen Überprüfung ob die Request-Parameter vordefinierten Bedingungen entsprechen (Process Validations). Sollte die Überprüfung nicht den Bedingungen entsprechen, wird die Verarbeitung abgebrochen. Wenn die Parameter in Ordnung sind, wird das Datenmodel aktualisiert (Update Model Values). Sollte bei einer Konvertierung ein Fehler auftreten wird die Verarbeitung abgebrochen. Danach wird die "Business Logik" abgearbeitet und die *in den Komponenten gespeicherten Daten* aktualisiert oder, wenn ein Seitenwechsel stattfinden soll, entschieden zu welchem View weitergeleitet werden soll (Invoke Application). Am Ende des Lifecycles wird immer dafür gesorgt, dass die Komponenten ihrem aktuellen Zustand entsprechend dargestellt werden (Render Response).

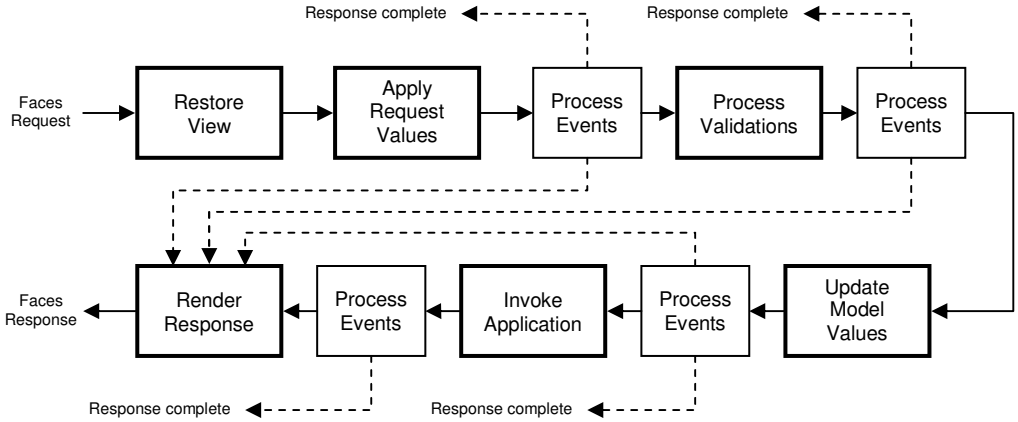

Abbildung 4-11
JavaServer Faces-Lifecycle

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>

<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
```

Beispiel 4-25
Ausschnitt aus `web.xml`

Welche Seiten der JSF-Applikation unter welchen Bedingungen auf welche anderen Seiten weiterleiten wird als Navigationsregeln in `faces-config.xml` definiert (=Faces-Requests von Faces-Seiten). In den Komponenten-Tags in den JSP-Seiten sind keine Verweise enthalten. HTML-Links führen aus der JSF-Applikation heraus (= Non-Faces-Request).
Mit dem Tag `<navigation-rule>` wird dem `NavigationHandler` mitgeteilt von welchem View `<from-view-id>` ein welchen Fällen `<navigation-case>` unter welcher Bedingung

`<from-outcome>` wohin `<to-view-id>` weitergeleitet werden soll. Dies muss für alle Seiten, von denen auf ein anderes View weitergeleitet werden soll, definiert werden.

```
<?xml version="1.0"?>

<!DOCTYPE faces-config
    PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  <navigation-rule>
    <from-view-id>/aufgabenstellung.jsp</from-view-id>
    <navigation-case>
      <from-outcome>richtig</from-outcome>
      <to-view-id>/richtig.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>falsch</from-outcome>
      <to-view-id>/falsch.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/richtig.jsp</from-view-id>
    <navigation-case>
      <from-outcome>neu</from-outcome>
      <to-view-id>/aufgabenstellung.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/falsch.jsp</from-view-id>
    <navigation-case>
      <from-outcome>neu</from-outcome>
      <to-view-id>/aufgabenstellung.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>


  ...

</faces-config>
```

Beispiel 4-26
Navigationsregeln in `faces-config.xml`

Es sei hier betont, dass im Gegensatz zu den bisherigen Konzepten in JSF das Klicken auf einen Link nicht unbedingt auf eine andere Seite verweisen muss oder zur Darstellung einer anderen Seite führt. Es kann ein Link auch eine Komponentenänderung hervorrufen, was bewirkt, dass die gleiche Seite erneut, aber mit verändertem Zustand der enthaltenen Komponenten zur Darstellung kommt.

Bei JSF werden die Beans, die als Model oder zur Kommunikation zwischen Model und den View-Komponenten dienen, als *Managed Beans* in `faces-config.xml` deklariert. Dadurch werden die Beans mit dem JSF-Framework verbunden und die Kontrolle dem Framework übergeben. Als Vorteil ist hier vor allem zu nennen, dass die Deklaration der verwendeten Beans an zentraler Stelle erfolgt und sie nicht über die ganze Applikation verstreut sind. Man kann auch Initialisierungswerte zentral definieren. Dies geschieht mit Hilfe der JSF-EL (JavaServerFaces-Expression Language).
Die JSF-EL kommt auch in JSF-Tags in den JSP-Seiten zum Einsatz um die Eigenschaften der Beans und die Komponenten miteinander zu verbinden. Sie unterscheidet sich von der JSP/JSTL-EL vor allem durch ihre leicht veränderte Syntax. Statt eines vorangestellten $-Zeichens ist hier ein #-Zeichen dem in geschwungene Klammern eingefassten Ausdruck vorangestellt: `#{`*Ausdruck*`}`
Es können die von der JSP-EL bekannten impliziten EL-Objekte (z.B. `applicationScope`) zum Zugriff auf implizite JSP-Objekte oder Initialisierungsparameter (`initParam`) verwendet werden.
Es können auch Abhängigkeiten der Managed Beans untereinander definiert werden. Im Beispielprogramm unten wird dies genützt damit die Managed Bean `Testverlauf` die Referenz auf die Managed Bean `Aufgabenkatalog` erhält: der Eigenschaft `katalog` von `Testverlauf` wird der Wert der Eigenschaft `referenz` von `Aufgabenkatalog` zugewiesen. Dadurch wird jedes neue `Testverlauf`-Objekt automatisch mit dem Zeiger auf `Aufgabenkatalog` initialisiert.

```
  <managed-bean>
    <managed-bean-name>Aufgabenkatalog</managed-bean-name>
    <managed-bean-class>
      at.ac.akhwien.mvc.model.Aufgabenkatalog
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
      <property-name>dateiname</property-name>
      <value>#{applicationScope.dateiname}</value>
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Testverlauf</managed-bean-name>
    <managed-bean-class>
      at.ac.akhwien.mvc.model.Testverlauf
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>testumfang</property-name>
      <value>#{initParam.testumfang}</value>
    </managed-property>
    <managed-property>
      <property-name>katalog</property-name>
      <value>#{Aufgabenkatalog.referenz}</value>
    </managed-property>
  </managed-bean>

</faces-config>
```

Beispiel 4-27
Ausschnitt aus `faces-config.xml`

Bei den Beans werden *Managed-Beans* und *Backing-Beans* unterschieden, wobei eine Backing Bean genauso als `<managed-bean>` deklariert wird.

- Managed-Beans enthalten die Business-Logik und sind für den Zugriff auf das Datenmodell zuständig.
- Backing-Beans stellen die Verbindung mit den UI-Komponenten und können auch die *Action-Methoden* enthalten, die für die Verarbeitung von Aktionen verantwortlich sind.

Demnach kann man die Bean `Aufgabenkatalog` als Managed-Bean und die Bean `Testverlauf` als Backing-Bean sehen.

Da das `FacesServlet` nur gestartet wird wenn im URL nach der Referenz auf die Web-Applikation `/faces/` folgt, muss der Zugriff auf die Index-Seite auf die Faces-Index-Seite umgeleitet werden.

```
<html>
  <head>
    <meta http-equiv="Refresh"
          content="0;URL=faces/index.jsp" />
  </head>
  <body bgcolor=white>
    Weiterleitung ...
  </body>
</html>
```

Beispiel 4-28
Ausschnitt aus `index.html`

Die Dateien `index.html` und `index.jsp` liegen im selben Verzeichnis (im Wurzelverzeichnis der Web-Applikation). Würde auf `/index.jsp` zugegriffen werden (ohne `/faces/` vorangestellt), dann würde die Seite ihrerseits auf die Seite `/aufgabenstellung.jsp` verlinken und nicht auf `/faces/aufgabenstellung.jsp`. Bei einem Klick auf den Link würde der WebServer versuchen die in `aufgabenstellung.jsp` enthaltenen JSF-Tags abzuarbeiten, aber daran scheitern und mit einer Fehlermeldung antworten, da das `FacesServlet` nicht gestartet wurde und die für die Komponenten-Tags notwendigen Vorarbeiten nicht erledigt wurden:

```
HTTP Status 500
The server encountered an internal error () that prevented it from
fulfilling this request.
javax.servlet.ServletException: Cannot find FacesContext
```

Auch wenn `index.jsp` selbst keine Faces-Komponenten enthält, wird durch den Aufruf von `/faces/index.jsp` das `FacesServlet` gestartet, das seinerseits wieder auf `index.jsp` weiterleiten wird, damit die Seite zur Darstellung kommt.

```
<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabenstellung.jsp">Start</a>.
```

```
    </hr>

  </body>
</html>
```

Beispiel 4-29
Ausschnitt aus `index.jsp`

Damit automatisch die richtige index-Seite vom Server geliefert wird, muss in `web.xml` die Datei `index.html` angegeben werden.

```
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
```

Beispiel 4-30
Ausschnitt aus `web.xml`

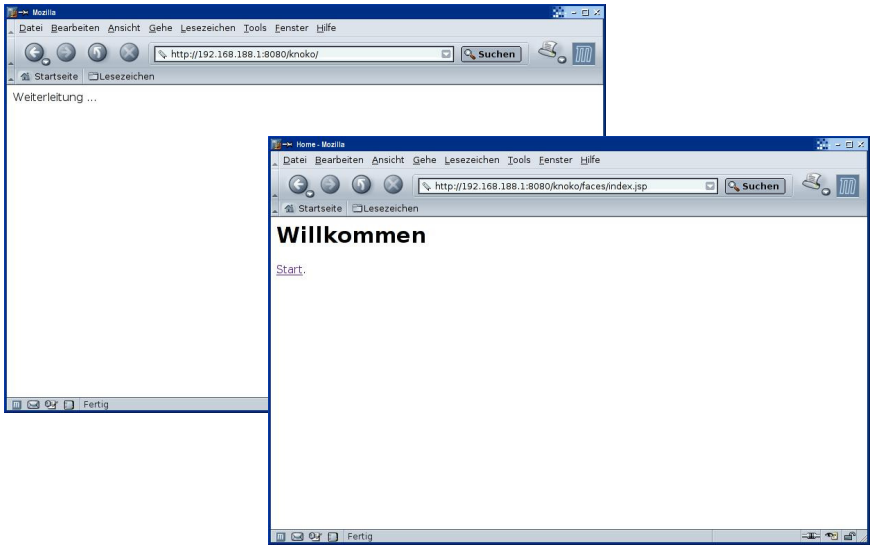

Abbildung 4-12
Screenshots von der Weiterleitung zu `index.jsp`

Wird von den Komponenten in den JSPs auf die Beans zugegriffen müssen die Beans nicht wie üblich mittels `<jsp:useBean>`-Aktion explizit deklariert werden. Die Beans werden automatisch geladen sobald ein JSF-Tag darauf zugreifen will. JSF bedient sich hierbei des Konzepts des *lazy loadings*. Die Beans werden erst dann geladen wenn sie von JSF benutzt werden. Man kann zwar mit den von JSP/JSTL bekannten Mechanismen auf die Beans einer JSF zugreifen, aber es ist zu beachten, dass auf eine Bean z.B. mit einem JSTL-Tag erst dann zugegriffen werden kann, nachdem sie von einem JSF-Tag benutzt wurde.

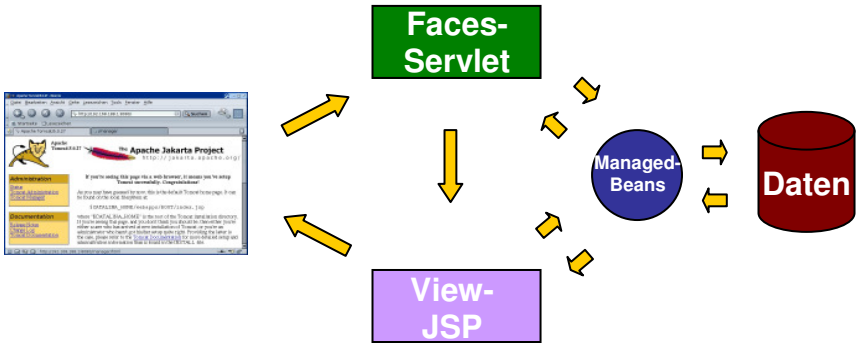

Abbildung 4-13
Beispiel-Komponenten für Ereignisverarbeitung

Die JSF-Referenz-Implementation stellt Core-Tags und HTML-Tags bereit.

Core-Tag-Library

Die Core-Tags (Präfix: `f`) werden für die grundlegende JSF-Funktionalität benötigt. So müssen alle JSF-Tags einer Seite im Rumpf des `<f:view>`-Tags stehen. Weiters stellt die Tag-Library unter anderem Tags zur Konvertierung (`<f:convertNumber>`, …), Validierung (`<f:validateLength>`, …) und die Ereignisbearbeitung (`<f:actionListener>`, `<valueChangeListener>`) bereit.

HTML-Tag-Library

Die HTML-Tag-Library (Präfix: `h`) dient zur Darstellung („*Rendern*“) von HTML-Elementen. Sie umfasst View-Komponenten zur Datenausgabe (z.B. `<h:outputText>`, `<h:dataTable>`, `<h:message>`, …), Dateneingabe (`<h:form>`, `<h:inputText>`, `<h:selectManyListbox>`, …) und Komponenten für Links und Buttons (`<h:commandLink>`, `<h:commandButton>`).

Im Rahmen des MyFaces-Projektes wurden neben weiteren HTML-Komponenten-Tags auch WML-Komponenten (für WAP-Anwendungen) implementiert.

Wie schon oben erwähnt, kommt in den JSF-Tags der JSP-Seiten die JSF-EL zum Einsatz um die Eigenschaften der Beans und die Komponenten miteinander zu verbinden. Hierbei wird *Value-Binding* und *Method-Binding* unterschieden.

Value-Binding

Beim Value-Binding wird eine Eigenschaft mit einem UI-Komponenten über das value-Attribut verbunden Diese Verbindung gilt für beide Richtungen: nicht nur zur Ausgabe von Daten (z.B. mittels `<h:outputText >`) sondern auch zur Ein- und Ausgabe von Daten (über z.B. `<h:inputText>`).

Method-Binding

Mit den Tags `<h:commandLink>` und `<h:commandButton>` wird auf andere Seiten weitergeleitet. Der Navigation-Handler des FacesServlets erledigt dies abhängig vom im Attribut `action` enthaltenen Wert und den in faces-config.xml definierten Navigationsregeln. Der Wert des Attributes `action` kann fix vorgegeben sein (= *statische Navigation*) oder von

der Auswertung einer *Action-Methode* abhängig sein (= *dynamische Navigation*). Wenn eine *Action-Methode* ausgewertet wird, nennt man das Method-Binding.

In der JSP `aufgabenstellung.jsp` wird das JSTL-Core-Tag-Konstrukt `<c:choose>-<c:when>` verwendet um, abhängig vom Fragentyp, das richtige Layout darzustellen. Während in den JSF-Tags die JSF-EL Anwendung findet, wird in den JSTL-Tags die JSP/JSTL-EL eingesetzt.
Außer `<html>` und `<body>` kommen keine HTML-Tags zum Einsatz. Die Seite wird nur mit Hilfe von HTML-Komponenten und CSS-Definitionen aufgebaut.

Um die Benutzereingabe auswerten zu können, werden die `<selectOneRadio>`-Tags über Value-Binding mit der Eigenschaft `antwort` des Objektes `Testverlauf` verbunden.

Die Auswertung ob auf richtig oder falsch geschieht durch dynamische Navigation mit einem Method-Binding-Ausdruck. Der Link zur Beendung des Tests wurde als statische Navigation realisiert.

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>



<html>
  <body>

    <f:view>

      <h:form>
        <h:outputText value="Fragennummer #{Testverlauf.aufgabe.nr}, " />
        <c:choose>
          <c:when test="${Testverlauf.aufgabe.modus==1}">
            <h:outputText value="Einfachauswahl (A)" />
            <h:panelGrid columns="1" styleClass="strich">

              <h:panelGrid columns="2"
                           columnClasses="links,rechts" >
                <h:outputText value="#{Testverlauf.aufgabe.fragenintro}"
                              escape="false" />
                <c:choose>
                  <c:when test="${Testverlauf.aufgabe.bildname!=''}">
                    <h:graphicImage value="#{Testverlauf.aufgabe.bildname}" />
                  </c:when >
                </c:choose>
              </h:panelGrid >

              <h:selectOneRadio id="wahlantworten"
                                value="#{Testverlauf.antwort}"
                                layout="pageDirection">
                <f:selectItem itemValue="A"
                         itemLabel="A .. #{Testverlauf.aufgabe.wahlantwort1}" />
                <f:selectItem itemValue="B"
                         itemLabel="B .. #{Testverlauf.aufgabe.wahlantwort2}" />
                <f:selectItem itemValue="C"
                         itemLabel="C .. #{Testverlauf.aufgabe.wahlantwort3}" />
                <f:selectItem itemValue="D"
                         itemLabel="D .. #{Testverlauf.aufgabe.wahlantwort4}" />
                <f:selectItem itemValue="E"
                         itemLabel="E .. #{Testverlauf.aufgabe.wahlantwort5}" />
              </h:selectOneRadio>
            </h:panelGrid >
```

```
          </c:when >

          <c:when test="${Testverlauf.aufgabe.modus==2}">
            <h:outputText value="Antwortkombinationsaufgabe (Kprim)" />

              ...

            </h:panelGrid>
          </c:when >

          <c:when test="${Testverlauf.aufgabe.modus==3}">
            <h:outputText value="Zuordnungsfrage (B)" />

              ...

          </c:when >
        </c:choose>

        <h:commandButton action="#{Testverlauf.aufgabekorrigieren}"
                         value="Abschicken" />
      </h:form>

      <h:panelGrid columns="1" styleClass="strich">
        <h:panelGroup>
          <h:outputText value="Zeit seit Beginn #{Testverlauf.dauer} sec. " />
          <h:outputText value="Es wurden bisher #{Testverlauf.gestellteAufgaben}
                               Fragen gestellt, " />
          <h:outputText value="davon richtig: #{Testverlauf.richtigeAufgaben}"/>
        </h:panelGroup>
      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="strich" columnClasses="rechts">
          <h:commandLink action="ende">
            <h:outputText value="Test vorzeitig beenden." />
          </h:commandLink>
        </h:panelGrid >
      </h:form>

    </f:view>

  </body>
</html>
```

Beispiel 4-31
Ausschnitt aus `aufgabenstellung.jsp`

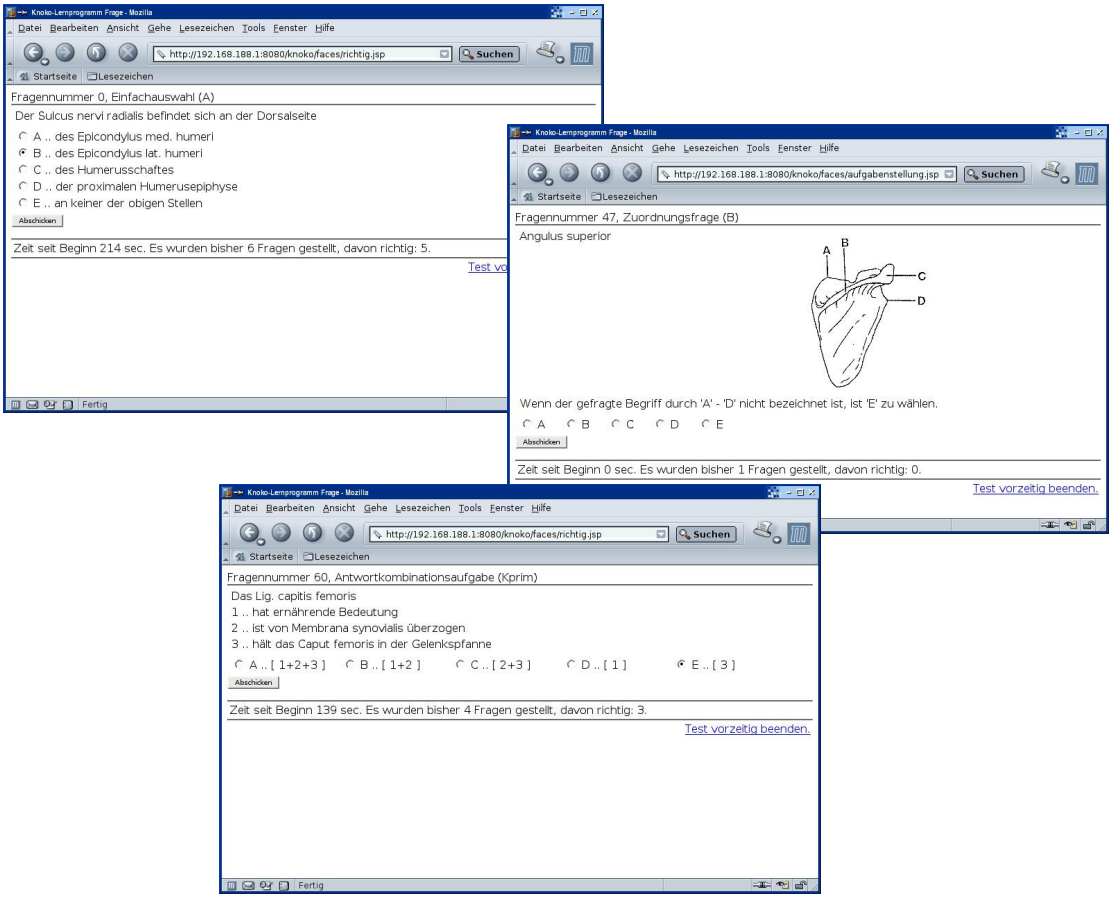

Abbildung 4-14
Screenshots der drei Fragentypen (JSF)

Der `RenderKit` erzeugt aus den `<h:commandLink>`-Tags HTML-Links, die auf "#" zeigen. Für den Eigentlichen Aufruf und die Parameterübergabe wird JavaScript im `onClick`-Tag verwendet. Das Formular mit dem Link zum Beenden ergibt von JSF gerendert folgenden HTML-Code:

```
<form id="_id20" method="post"
      action="/knoko/faces/aufgabenstellung.jsp"
      enctype="application/x-www-form-urlencoded">
  <table class="strich">
    <tbody>
      <tr>
        <td class="rechts">
          <a href="#" onclick="document.forms['_id20']['_id20:_idcl'].va
lue='_id20:_id22'; document.forms['_id20'].submit(); return false;">Test
vorzeitig beenden.</a>
        </td>
      </tr>
    </tbody>
  </table>
  <input type="hidden" name="_id20" value="_id20" />
  <input type="hidden" name="_id20:_idcl" />
</form>
```

Beispiel 4-32
von JSF generierter HTML-Code aus `aufgabenstellung.jsp`

Die Seite `richtig.jsp` verwendet eine Checkbox-Komponente und definiert mittels `value`-Attribute, dass der vom Benutzer eingegebene Parameter der Eigenschaft aufgabeWiederholen von Testverlauf zugewiesen werden soll.
Der `<commandButton >` wird an die Methode neueAufgabe verbunden. Die Weiterleitungen in Abhängigkeit des Rückgabewertes sind wieder mittels Navigationsregeln in `faces-config.xml` definiert. Dort ist auch die Seite für die fix vorgegebene `action` des `<commandLink>`-Tags vereinbart.

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>

<html>
  <body>

    <f:view>

      <h:outputText value="Fragennummer #{Testverlauf.aufgabe.nr}" />

      <h:panelGrid columns="1" styleClass="alles" columnClasses="strich">
        <h:panelGroup>
          <h:outputText styleClass="h1" value="Richtig!" />

          <h:form>
            <h:panelGrid columns="1" columnClasses="doppelt" >
              <h:outputText value="Gegebene Antwort: #{Testverlauf.antwort},
                              richtige Antwort: #{Testverlauf.aufgabe.loesung}" />
              <h:panelGroup>
                <h:selectBooleanCheckbox id="wiederholen"
                                         value="#{Testverlauf.aufgabeWiederholen}" />
                <h:outputLabel for="wiederholen">
                  <h:outputText value="diese Aufgabe nocheinmal stellen" />
                </h:outputLabel>
              </h:panelGroup>

              <h:commandButton action="#{Testverlauf.neueAufgabe}"
                               value="Weiter" />
            </h:panelGrid >
          </h:form>
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt" >
          <h:outputText value="Zeit seit Beginn:
                               #{Testverlauf.dauer} Sekunde(n)." />
          <h:outputText value="Es wurden bisher #{Testverlauf.gestellteAufgaben}
                               Fragen gestellt, " />
          <h:outputText value="davon richtig: #{Testverlauf.richtigeAufgaben}"/>
        </h:panelGroup>

      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="alles" rowClasses="strich"
                                                    columnClasses="rechts">
          <h:commandLink action="ende">
            <h:outputText value="Test vorzeitig beenden." />
          </h:commandLink>
        </h:panelGrid >
      </h:form>

    </f:view>
  </body>
</html>
```

Beispiel 4-33
Ausschnitt aus `richtig.jsp`

Wie schon in den vorangegangenen MVC-Beispiel-Applikationen entspricht die Seite `falsch.jsp` größtenteils der Seite `richtig.jsp`. Hier unterscheiden sich die Seiten nur dadurch, dass in `falsch.jsp` „Falsch!“ ausgegeben wird. Ob die Checkbox angehackt ist, wird in der Bean Testverlauf entschieden – die Methode `getAufgabeWiederholen()` liefert wenn das Ergebnis richtig ist den Wert "`false`" und sonst "`true`".
Da die Komponente `<selectBooleanCheckbox>` und die Eigenschaft `aufgabeWiederholen` über ValueBinding verbunden sind, holt sich die Komponente am Anfang des JSF-Lifecycles den Wert der Eigenschaft und wird entsprechend gerendert. Wird der Wert geändert wird er vom Server als Parameter an das FacesServlet gesendet und die Eigenschaft aktualisiert.

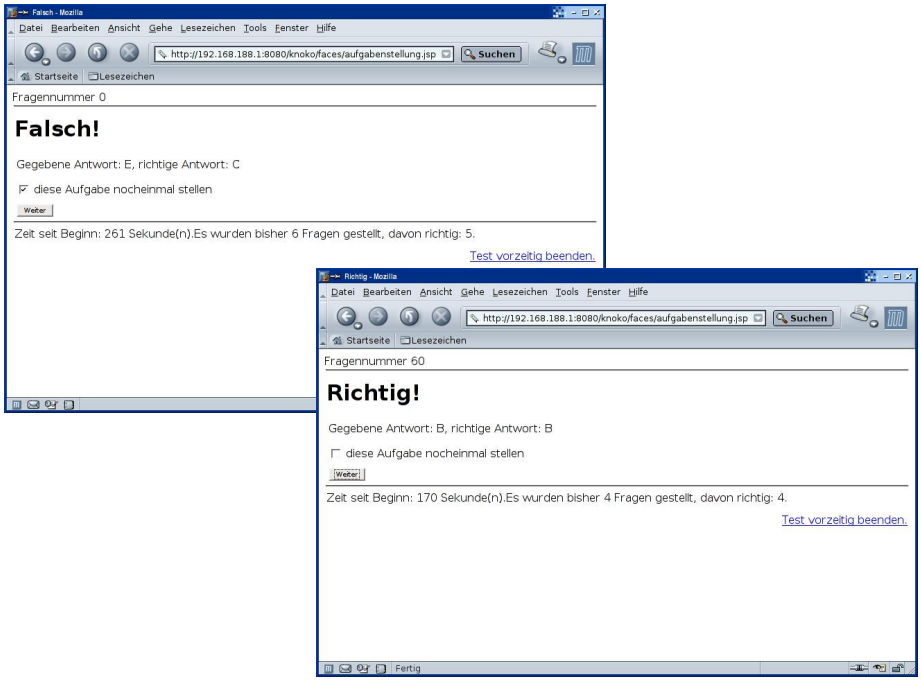

Abbildung 4-15
Screenshots von `richtig.jsp` *und* `falsch.jsp` *(JSF)*

Die JSF-EL bietet die gleichen Operationen wie die JSP/JSTL-EL um Rechenoperationen für die Statistik auf die Eigenschaften anzuwenden.
Das JSF-Framework kümmert sich durch das Managed-Beans-Konzept selbständig darum welche Objekte automatisch in den Session-Scope geladen werden müssen, jedoch muss die Session trotzdem explizit beendet werden. Auch hier muss darauf geachtet werden, dass die Session erst am Ende der Seite beendet wird, wenn auf Testverlauf nicht mehr zugegriffen wird. Mit dem JSF-Core-Tag `<f:convertNumber maxFractionDigits="2" />` werden die überzähligen Kommastellen abgeschnitten.

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>

<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>

<html>
  <body>

    <f:view>

      <h:panelGrid columns="1" >
        <h:outputText value="Ende" styleClass="h1" />
```

```
        <h:outputText value="Auswertung:" styleClass="h2" />

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Vom Computer wurden #{Testverlauf.testumfang}
                               verschieden Fragen " />
          <h:outputText value="aus einem Pool von #{Testverlauf.katalog.anzahl}
                               Aufgaben per Zufall ausgesucht." />
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Die Dauer zur Beantwortung aller Fragen betrug
                               #{Testverlauf.dauer} Sekunde(n). " />
          <h:outputText value="Dies entspricht "/>
          <h:outputText
                    value="#{Testverlauf.dauer/Testverlauf.gestellteAufgaben}">
            <f:convertNumber maxFractionDigits="2" />
          </h:outputText >
          <h:outputText value=" Sekunden pro Frage." />
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Insgesamt wurden#{Testverlauf.gestellteAufgaben}
                                                          Fragen gestellt, " />
          <h:outputText value="von denen Sie #{Testverlauf.richtigeAufgaben}
                                                  richtig beantwortet haben ( " />
          <h:outputText value="#{ 100*Testverlauf.richtigeAufgaben /
                                                  Testverlauf.gestellteAufgaben}">
            <f:convertNumber maxFractionDigits="2" />
          </h:outputText >
          <h:outputText value=" % )." />
        </h:panelGroup>

      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="strich" >
          <h:outputLink value="index.jsp" >
            <h:outputText value="zum Start" />
          </h:outputLink>
        </h:panelGrid >
      </h:form>

    </f:view>

  </body>
</html>

<sess:invalidate/>
```

Beispiel 4-34
Ausschnitt aus `ende.jsp`

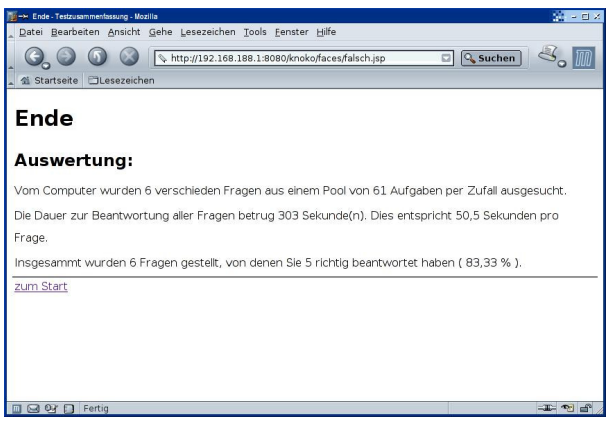

Abbildung 4-16
Screenshot von `ende.jsp` *(JSF)*

5. Diskussion

Alle drei Varianten ermöglichen in Model, View und Controller einen gut lesbaren Code zu entwickeln.

Die durch Einsatz des MVC-Konzepts erhoffte Trennung von Darstellung und Code ist differenziert zu sehen.

5.1. Allgemeines

Performance

Ein Vergleich der Leistungsfähigkeit bei der Auslieferung von dynamischen Web-Seiten [53] ergab, dass eine JavaServer basierte Lösung durchaus mit Maschinenabhängigem C-Code mithalten kann. Die Messungen wurden mit ab2 (Apache Bench 2) durchgeführt und die Aufgabe bestand, darin dass 100 bzw.1000 Zeilen pro Request vom Server geliefert werden mussten. Die Performance wurde in Bezug auf die Auslieferungsleistung für eine statische Seite mit 100 bzw. 1000 Zeilen angegeben. Es wurden hierbei die Skriptsprache PHP, in C geschriebenes CGI und FastCGI, Servlet mit Velocity (ein Template-Framework), JSP, JSP mit JSTL und Servlets (optimiert und nicht optimiert) miteinander verglichen. Es zeigte sich überraschenderweise, dass reine Servlets bei den kleinen Dokumenten schneller waren als die C-CGI-Programme und über 70% der Performance im Vergleich zum statischen Dokument erreichten.
Weiters wurde deutlich, dass Servlets schneller als JSPs und diese wiederum schneller als JSPs mit JSTL waren. Dies war zu erwarten, da die JSTL aufgrund ihrer Universalität einen dementsprechenden Overhead zu bearbeiten hat, der für diesen synthetischen Test nicht benötigt wird. Die JSTL-Version erreichte in diesem Test nur ca. 3% im Vergleich zur statischen Seite bei großen Dokumenten.

Im Rahmen des Vergleichs der MVC-Konzepte wurden keinerlei Leistungsmessungen vorgenommen. Es war aber deutlich zu bemerken, dass die Dauer für die erste Bereitstellung (d.h. inklusive Compilierung der JSP) bei JSP/JSTL-Version etwa genauso lang war wie bei Struts, die Seiten der JSF-Version jedoch wesentlich länger auf sich warten ließen.

Einen ungefähren Anhaltspunkt kann ein Vergleich der Dateigrößen der Class-Dateien geben. Hierzu wurden die von Tomcat 5 erstellten class-Dateien aus dem work-Verzeichnis und die vorcompilierten Class-Dateien der Beans und Servlets im `/WEB-INF/lib`-Verzeichnis gegenübergestellt.

Bytecode		JSTL	Struts	JSF
generiert	[Bytes]	68.312	57.758	103.521
vorcompiliert	[Bytes]	11.622	17.236	10.948
Summe	[Bytes]	79.934	74.994	114.469
Prozent	%	100	94	143
Model-Bytecode	[Bytes]	10.948	10.948	10.948
Summe-Model	[Bytes]	68.986	64.046	103.521
Prozent	%	**100**	**93**	**150**

Tabelle 5-1
Bytecode-Vergleich

Quellcode		JSTL	Struts	JSF
JSP-Code	[Bytes]	14.388	12.374	14.395
Prozent	%	**100**	**86**	**100**

Tabelle 5-2
Quellcode-Vergleich

Es zeigt sich, dass die JSF-Version um die Hälfte mehr Bytecode umfasst als die JSTL-Version (sowohl inklusive als auch exklusive Model-Code). Der Vergleich des Quellcodes ist nicht als Vergleichsmaßstab zu gebrauchen.

Dies legt nahe, dass damit gerechnet werden muss, dass ein Server der JSF-Applikationen im Internet bereitstellt, eine höhere Servlerlast verkraften können muss.

Features

Die hier behandelten Ansätze zur Programmierung einer WWW-Applikation in Java unterstützen Programmierer und Web-Designer in verschiedenem Maße bei der Entwicklung .

	Servlets	JSP	JSTL	Struts	JSF
Autom. Formularüberprüfung	-	-	-	++	++
MVC Trennung „by Design“	-	+/-	+/-	++	++
IDE support / graph. UI-Design	+/-	++/-	++	+	++
Bean Unterstützung	+	+	+	++	++
I18n	-	+/-	+	+	++
UI-Komponenten	-	-	-	-	++
UI-Events	-	-	-	-	++

Tabelle 5-3
Feature-Vergleich

5.2. Model

Durch die Auslagerung in von Code in Beans lässt sich zweifellos der Java-Code-Anteil in den JSPs reduzieren bzw. zur Gänze vermeiden. Bei der Entwicklung von Beans sollte man sich nicht auf die Speicherung von Eigenschaften und den Zugriff auf diese Eigenschaften über die `set-` und `get-`Methoden beschränken, sondern unterstützende Funktionen, die die Eigenschaften des Objekts betreffen, ebenfalls in dem Objekt kapseln und als öffentliche Methoden für die Arbeit mit dem Objekt zur Verfügung zu stellen.

Da das JSF-Beispiel das erste war, das programmiert wurde, ist die Entwicklung der Model-Beans durch die speziellen Bedürfnisse von JSF beeinflusst worden – einerseits um eine funktionsfähige JSF-Applikation erstellen zu können, andererseits um die Möglichkeiten von JSF zeigen zu können. Es muss damit gerechnet werden, dass Beans die schon in anderen Java-Projekten im Einsatz sind (stand-alone Java-Applikationen, Applets oder Web-Applikationen die mittels Servlets, JSPs oder Struts realisiert sind), sich nicht ohne Anpassung als Managed-Beans verwenden lassen. Es werden daher zusätzliche Wrapper-Objekte notwendig sein, die den Zugriff auf die Objekte vermitteln und die notwendigen Action-Methoden implementieren.

Bei Struts sind die Form-Beans von ActionForm abgeleitet. Form-Beans müssen daher theoretisch immer zusätzlich zu den schon vorhandenen Model-Beans erstellt werden. Seit Version 1.1 bietet Struts dynamische Form-Beans, die vom Framework automatisch erstellt werden und somit die explizite Programmierung von Form-Beans in vielen Fällen unnötig machen. Einen Vorteil bietet das Form-Beans-Konzept vor allem bei Formularen, die über mehrere Web-Seiten verteilt sind: wenn alle Formular-Seiten ausgefüllt und abgeschickt sind befinden sich alle eingegeben Daten in nur einer Bean.

Alle drei Varianten haben die Methoden `neueAufgabe()` und `aufgabekorrigieren()` der Bean `Testverlauf` gut nützen können. Wären in diese Methoden nicht implementiert, würde dies für die JSP/JSTL- und die Struts-Version keine Probleme bedeuten. Die Überprüfung könnte genauso durch einen Vergleich des Parameters `antwort` mit der Eigenschaft `loesung` erfolgen. Bei der JSP/JSTL-Version durch ein `<c:choose>-<c:when>`-Konstrukt, bei Struts durch eine einfache if-else-Abfrage in der Action-Klasse. Da JSP keine direkte Unterstützung im Sinne eines generischen Tags für den Aufruf von Bean-Methoden[54] bietet, würde es in der JSP/JSTL-Version die Programmierung sogar vereinfachen, weil auf die Programmierung der Funktionen verzichtet werden könnte.

```
<c:choose>
  <c:when test="${testverlauf.aufgabe.loesung == testverlauf.antwort}">
    <jsp:forward page="richtig.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="falsch.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 5-1
Ausschnitt aus Controller-JSP `korrektur.jsp`

Die JSF-Version nützt die Bean Methoden sehr elegant als Action-Methoden. Wären sie nicht vorhanden könnte man sich mit einem EL-Ausdruck behelfen

```
      <h:commandButton
               action="#{Testverlauf.aufgabe.loesung == Testverlauf.antwort}"
               value="Abschicken" />
```

Beispiel 5-2
Ausschnitt aus der JSF-Seite `aufgabenstellung.jsp`

Der Rückgabewert wäre dann aber entweder "true" oder "false". Somit müssen auch die Navigationsregeln angepasst werden. Allerdings dürften dann nicht mehr als eine Überprüfung mit EL-Ausdrücken in den Action-Methoden der Seite stattfinden.

```
  <navigation-rule>
    <from-view-id>/aufgabenstellung.jsp</from-view-id>
    <navigation-case>
      <from-outcome>true</from-outcome>
      <to-view-id>/richtig.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>false</from-outcome>
      <to-view-id>/falsch.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
```

```
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

Beispiel 5-3
Navigationsregeln in `faces-config.xml`

Die Lösung über die JSTL mit `<c:set>` eine Variable innerhalb eines ein `<c:choose>`-`<c:when>`-Konstrukts zu setzen und diese dann dem action-Attribut zu übergeben wäre zwar denkbar, widerspräche jedoch gänzlich dem Design von JSF.
Eine weitere Möglichkeit wäre einen `ActionListener` zu implementieren.

5.3. Controller

Wo die Navigationslogik, ActionListener oder Action-Methoden implementiert werden sollen gibt die JSF-Spezifikation nicht vor. Die Auswertung und Weiterleitung findet zwar im FacesServlet statt, da es aber sich bei JSF um eine seiten-orientierte Architektur handelt, wird kein Controller im Sinne von Struts programmiert, der über die Weiterleitung entscheidet. Es ist daher beim Schreiben der Beans darauf zu achten, dass der Model-Code nicht mit JSF-spezifischem Code vermischt wird, damit sie wiederverwendbar bleiben.

Seit der Einführung der JSF stellt sich die Frage ob dadurch das etablierte und weit verbreitete Struts-Framework noch weiterentwickelt wird [55]. Die Struts-Entwickler haben dazu vor einiger Zeit eine Feststellungserklärung veröffentlicht, die besagt, dass sie JSF und Struts als komplementäre Technologien sehen und Struts in Hinblick auf eine bessere Unterstützung der JSF weiterentwickelt wird. Das Struts-Framework wurde vor einiger Zeit im Rahmen auf die Umstellung auf die modularere neue Version in mehrere Sub-Projekte aufgeteilt. Andererseits wurde angekündigt, dass es zwei Hauptstränge der Entwicklung entstehen werden. Das *Struts Action Framework* als Nachfolger des bekannten Struts Frameworks und das *Struts Shale Framework* [56], als ein komponenten-orientiertes Framework basierend auf der JSF-Technologie. Shale stellt dazu unter anderem seiten-orientierte View-Controller und den für Struts typischen Controller ("Application-Manager") zur Verfügung. Das Struts Shale Framework wurde vor kurzem zum Apache Top-Level-Projekt und in stabiler Version veröffentlicht. Die Version 2 von Struts steht kurz vor der Veröffentlichung.

Der Programm-Logik, die bei JSF und Struts entscheidet unter welchen Bedingungen weitergeleitet wird, ist als Java-Code in Beans bzw. Servlets programmiert. Das bedeutet, dass wenn Änderungen vorgenommen werden müssen, dann muss der Quellcode verfügbar sein und nach Adaptierung neu compiliert werden. Die JSTL-Version bietet den Vorteil, dass die Controller jederzeit einfach mit einem normalen Text-Editor angepasst werden können.

Um nicht zu viele Controller warten zu müssen, umgehen manche Programmierer bei ihren Webapplikationen das Problem indem sie nur einen Controller besitzen, der mehrere Funktionalitäten erfüllen und auf den von den Views immer zurückverwiesen wird. Was der Controller zu tun hat wird ihm mittels Parametern übergeben. Dies schafft jedoch einen “Central Point of Failure”. Dessen sollte man sich vor allem beim Design großer Webapplikations-Projekte bewusst sein. Weiters wird dadurch eine sinnvolle *physikalische Navigation* (bei der von einer Seite zur nächsten wird) nicht mehr möglich.

5.4. View

Tags und EL

Die Beispiele zeigen aber auch, dass die Auslagerung von Code in Beans allein das Problem nicht löst.

Das Struts-Beispiel beschränkte sich auf den Einsatz der Struts-Tag-Library und verzichtete auf den Einsatz von JSTL und EL, um die Möglichkeiten bzw. Beschränkungen besser illustrieren zu können. Durch die eingeschränkten Möglichkeiten der Struts-Tag-Library war ein Einsatz von Scriptlets unumgänglich. Das bedeutet, dass vor allem die JSTL-Tags und die Expression Language einen großen Beitrag zur Reduktion des Java-Codes in den JSP-Seiten leisten. Andererseits jedoch handelt es sich bei den Tags und der EL im Grunde nur um eine andere Art von Programmcode und damit kommt es erneut zu einer von Vermischung von Darstellung und Code. Allerdings ist zu bedenken, dass ein Code, der nur aus Tags besteht und sich auf einfache Funktionen der Programmablaufsteuerung beschränkt, wesentlich leichter lesbar und wartbar ist. Selbst Web-Designer mit nur geringen Programmierkenntnissen sollten in der Lage sein JSTL-Seiten zu verstehen und eventuell sogar zu warten.

Darstellung

In der JSP/JSTL- und der Struts-Version können Web-Designer alle Möglichkeiten von HTML und CSS wie bei normalen Web-Seiten nützen. Bei den JSF kann (und sollte) im Idealfall gänzlich auf HTML-Code verzichtet werden. Das Layout sollte ausschließlich auf CSS basieren. Bei dem JSF-Beispiel zeigte sich, dass die Generierung der HTML-Tags JSF im Renderer durch seine Programmierung den Layout-Möglichkeiten Grenzen setzt. Das oben vorgestellte und geforderte Layout für den Fragentyp Kprim (zusätzlicher Label, der die Kombination der Wahlantworten unterhalb des Radio-Buttons zeigt) war mit JFS und CSS trotz zahlreicher Versuche nicht befriedigend zu realisieren.

Es existieren für alle 3 Technologien integrierte Entwicklungsumgebungen (IDEs), um Programmierer bei der Realisierung von Web-Applikationen zu unterstützen. Moderne Java-Entwicklungsumgebungen wie Borland JBuilder, Bea Workshop, Oracle JDeveloper, Sun Studio Creator, Sun NetBeans IDE oder das quelloffene Eclipse erleichtern die Programmierung von Servlets, JSPs, Struts und JSF . Von Seiten der professionellen Web-Design Applicationen unterstützt derzeit Adobe/Makromedia Dreamwaver neben PHP auch JSPs [57].

	Servlets	JSP	JSTL	Struts	JSF
JBuilder	+	+	+	+	+
Bea Workshop	+	+	+	+	+
JDevelop	+	+	+	+	+
Netbeans DIE	+	+	+	+	+
Studio Creator	+	+	+	+	+
Eclipse	+	+	+	+	+
Dreamwaver	-	+	+	-	-

Tabelle 5-4
IDE-Vergleich

JavaServer Faces-Seiten können nicht mit herkömmlichen HTML-Editoren erstellt werden. Einige Website-Entwicklungs-Tools wie Dreamweaver können mit Hilfe spezieller Extensions JSF-Seiten bearbeiten.
JavaServer Faces bietet zwar die Möglichkeit eigene Komponenten zu programmieren, jedoch ist dies mit einem noch viel höherem Aufwand verbunden als die Erstellung von Tags für eine benutzerdefinierte Tag-Library. Die Auswahl an Komponenten ist derzeit noch überschaubar gering und die Layout-Möglichkeiten sind trotz Einsatz von CSS aufgrund der vorgegebenen Eigenschaften und der Programmierung der Komponenten eingeschränkt.

Barierrefreiheit

In Deutschland wurde im Jahr 2002 die „*Barrierefreie Informationstechnik-Verordnung*“ (BITV)[58] als verbindliche Rechtsverordnung zu § 11 Behindertengleichstellungsgesetz des Bundes (BGG) basierend auf den Web Content Accessibility Guidelines 1.0 (WCAG1)[59] des World Wide Web Consortiums (W3C) verabschiedet.
Diese Verordnung stellt die Grundlage für barrierefreies Webdesign in Deutschland dar und enthält Anforderungen zur barrierefreien Gestaltung insbesondere von Webseiten der deutschen Bundesbehörden. Punkt 6 der Anforderungen besagt, dass Internetangebote auch dann nutzbar sein müssen, wenn der verwendete Benutzeragent neuere Technologien nicht unterstützt oder diese deaktiviert sind. Es wird gefordert, dass JavaScript vermieden werden soll oder zumindest ein Geräte-unabhängiger Event-Handler eingesetzt werden soll.
So erfordert der Eventhandler "`onClick`" zwingend einen Mausklick und kann nicht alternativ per Tastatur oder einem anderen Gerät ausgelöst werden. Die Tatsache, dass JSF den Tag `<f:commandLink>` in ein HTML-`<a>`-Tag umwandelt, das das Attribut `onClick` nutzt um zur nächsten Seite zu verlinken ist als negativ zu bewerten, da dadurch eine barrierefreies Webdesign unmöglich gemacht wird.

Internationalisierung (i18n)

Auf Internationalisierung wurde im Rahmen dieses Vergleichs kein Augenmerk gelegt. Alle drei Technologien erleichtern durch die Unterstützung von ResourceBundles die Erstellung mehrsprachiger Web-Applikationen. JSP/JSTL ermöglicht dies mit der Format-Tag-Library. Bei Struts und JSF kann man Platzhalter in länderspezifischen Dateien definieren und in den Tags anstelle von fixen Strings angeben

6. Zusammenfassung – Schlussfolgerungen – Ausblick

Es ist generell anzuraten eine Web-Applikation von Beginn an nach dem MVC-Konzept zu entwerfen. Alle 3 Technologien ermöglichen es gut lesbaren und wartbaren Code zu erzeugen, wenn sich der Programmierer an die Regeln der objektorientierten Programmierung hält und das Projektdesign von Beginn an dahingehend ausgelegt ist, dass die einzelnen Komponenten wiederverwendbar bleiben.

Auf welcher Technologie die Implementierung basieren soll ist nach den Anforderungen an die Web-Applikation zu entscheiden.

Die JavaServer Pages in Kombination mit der Java Standard Tag Library stellt eine verbreitete, stabile Basis für eine schnelle Entwicklung kleiner Projekte dar. Vor allem bei

Projekten, wo Personen die Seiten warten müssen, die nur rudimentäre HTML- und Programmierkenntnisse besitzen, empfiehlt sich der alleinige Einsatz von JSP/JSTL.

Größere Projekte sollten mit dem Struts (Action) Framework oder JavaServer Faces realisiert werden. Bestehende Struts-Projekte sollten nicht umgestellt werden. Struts ist ein etabliertes und stabiles Framework, das auf einem bewährten Konzept aufbaut und viele oft benötigte Funktionen zur Verfügung stellt.

Bei umfangreichen Projekten bietet JSF aufgrund seines Designs klare Vorteile. Es ist damit zu rechnen, dass sich die komponentenorientierte JSF-Architektur etablieren wird. Ob das seiten-orientierte JSF allein oder JSF in Kombination mit dem controller-orientierten Struts Shale Framework in Zukunft die bessere Wahl ist, wird sich zeigen, wenn Shale die notwendige Reife errreicht hat. Allerdings handelt es sich um ein Konzept das sowohl für Web-Designer noch Web-Applikations-Entwickler viel Neues bringt und daher mit dementsprechend viel Einarbeitungszeit verbunden ist. Bei einem engen Zeitplan und wenig Entwicklungserfahrung mit JSF sollte auf eine Technologie zurückgegriffen werden, mit der mehr Erfahrung besteht.

7. Verzeichnisse

Abbildungsverzeichnis

Abbildung 1-1 *Model 1* 4
Abbildung 1-2 *Model 2* 4
Abbildung 2-1 *WEB-INF-Verzeichnisstruktur* 8
Abbildung 2-2 *Servlet-Livecycle* 9
Abbildung 2-3 *Servlet-Container* 10
Abbildung 2-4 *gegenseitige Beeinflussung zweier Threads* 11
Abbildung 2-5 *Model 2 mit Servlets* 17
Abbildung 2-6 *Auflistung der im Session-Objekt gespeicherten Daten* 22
Abbildung 2-7 *MVC mittels JSPs* 59
Abbildung 3-1 *Flussdiagramm für Controller-Funktionen* 73
Abbildung 3-2 *Layouts der3 Fragentypen* 74
Abbildung 4-1 *Klassendiagramme der Model-Beans* 76
Abbildung 4-2 *MVC mittels JSP mit Zugiff auf Daten* 78
Abbildung 4-3 *Screenshots der drei Fragentypen (JSTL/JSP)* 82
Abbildung 4-4 *Screenshots von* `richtig.jsp` *und* `falsch.jsp` *(JSTL/JSP)* 84
Abbildung 4-5 *Screenshot von* `ende.jsp` *(JSTL/JSP)* 85
Abbildung 4-6 *MVC-Beispiel mit Stuts* 91
Abbildung 4-7 *Screenshots der drei Fragentypen (Struts)* 95
Abbildung 4-8 *Screenshots von* `richtig.jsp` *und* `falsch.jsp` *(Struts)* 98
Abbildung 4-9 *Screenshot von* `ende.jsp` *(Struts)* 99
Abbildung 4-10 *Beispiel-Komponenten für Ereignisverarbeitung* 101
Abbildung 4-11 *JavaServer Faces-Lifecycle* 103
Abbildung 4-12 *Screenshots von der Weiterleitung zu* `index.jsp` 107
Abbildung 4-13 *Beispiel-Komponenten für Ereignisverarbeitung* 108
Abbildung 4-14 *Screenshots der drei Fragentypen (JSF)* 111
Abbildung 4-15 *Screenshots von* `richtig.jsp` *und* `falsch.jsp` *(JSF)* 113
Abbildung 4-16 *Screenshot von* `ende.jsp` *(JSF)* 114

Tabellenverzeichnis

Tabelle 1-1 *Mitwirkende bei webbasierten Projekten* 4
Tabelle 2-1 *Implizite JSP-Objekte* 28
Tabelle 2-2 *Rückgabewerte von* `doStartTag()` *und* `doEndTag()` 40
Tabelle 2-3 *Rückgabewerte von* `doStartTag(), doInitBody(), doAfterBody` und `doEndTag()` 43
Tabelle 2-4 *Werte für scope und deren Gültigkeitsbereiche* 54
Tabelle 2-5 *EL-Operatoren* 63
Tabelle 2-6 *Implizite EL-Objekte* 63
Tabelle 2-7 *Scheifenstatus-Eigenschaften* 66
Tabelle 5-1 *Bytecode-Vergleich* 115
Tabelle 5-2 *Quellcode-Vergleich* 116
Tabelle 5-3 *Feature-Vergleich* 116
Tabelle 5-4 *IDE-Vergleich* 119

Beispiele

Beispiel 2-1 `ServletLivecycleDemo.java` *zur Illustration des Servlet-Livecycle* ... 11
Beispiel 2-2 *Thread-safe mittels* `synchronized()` ... 12
Beispiel 2-3 *HttpServlet-Beispiel* `Test.java` ... 13
Beispiel 2-4 *Definition des Servlet-Aufrufs in web.xml* ... 13
Beispiel 2-5 `input.html` ... 14
Beispiel 2-6 `output.html` ... 15
Beispiel 2-7 *RequestDispatcher-Servlet* `forward.java` ... 15
Beispiel 2-8 `outputheader.html` ... 16
Beispiel 2-9 `outputfooter.html` ... 16
Beispiel 2-10 *RequestDispatcher-Servlet* `include.java` ... 16
Beispiel 2-11 *RequestDispatcher-Servlet* `controller.java` ... 18
Beispiel 2-12 *RequestDispatcher-Servlet* `view.java` ... 19
Beispiel 2-13 *HTML-Header-Datei* `viewheader.html` ... 19
Beispiel 2-14 *HTML-Footer-Datei* `viewfooter.html` ... 19
Beispiel 2-15 *Session-Controller-Servlet* `sessioncontroller.java` ... 23
Beispiel 2-16 *Session-View-Servlet* `sessionview.java` ... 24
Beispiel 2-17 *Beispiel-JSP* `helloworld.jsp` ... 25
Beispiel 2-18 *Servlet-Code umgewandelte Beispiel-JSP* `helloworld.jsp` ... 26
Beispiel 2-19 *Beispiel-JSP* `scriptlet.jsp` ... 27
Beispiel 2-20 `impobj.jsp` ... 29
Beispiel 2-21 `expression.jsp` ... 29
Beispiel 2-22 `declarations.jsp` ... 30
Beispiel 2-23 `initdestroy.jsp` ... 31
Beispiel 2-24 `laufzeitfehler.jsp` ... 33
Beispiel 2-25 `fehlerseite.jsp` ... 33
Beispiel 2-26 `include.jsp` ... 34
Beispiel 2-27 `footer.html` ... 35
Beispiel 2-28 `test.jsp` ... 35
Beispiel 2-29 *vom JSP-Contrainer generierter Java-Servlet-Code* ... 36
Beispiel 2-30 `taglib.jsp` *mit Tag* `<bt:serverzeit>` ... 38
Beispiel 2-31 *Taglib-Einträge in* `/WEB-INF/web.xml` ... 38
Beispiel 2-32 *Taglib-Deskriptor* `bsptag.tld` ... 39
Beispiel 2-33 *Implementation des Tags mit TagSupport*: `BeispielTag.java` ... 41
Beispiel 2-34 *Implementation des Tags mit SimpleTagSupport*: `BeispielTag.java` ... 42
Beispiel 2-35 JSP `taglib.jsp` mit taglib-Direktive zum Aufruf der Beispiel Taglib ... 42
Beispiel 2-36 JSP `bspfunction.jsp` *mit JSP-Funktion* `${bt:serverzeit}` ... 45
Beispiel 2-37 *Taglib-Einträge in* `/WEB-INF/web.xml` ... 45
Beispiel 2-38 Die TagLibrary `bspfunction.tld` ... 46
Beispiel 2-39 Die Implementation der JSP-Funktion `bspfunction.java` ... 46
Beispiel 2-40 `input.html` ... 48
Beispiel 2-41 *Controller-JSP* `forward.jsp` ... 48
Beispiel 2-42 *JSP* `output.jsp` ... 49
Beispiel 2-43 `outputheader.html` ... 50

Beispiel 2-44 outputfooter.jsp 50
Beispiel 2-45 include.jsp 51
Beispiel 2-46 controller.jsp 51
Beispiel 2-47 view.jsp 52
Beispiel 2-48 BeispielBean.java 53
Beispiel 2-49 pagebean.jsp 55
Beispiel 2-50 pagebean_inc.jsp 56
Beispiel 2-51 requestbean.jsp 56
Beispiel 2-52 requestbean_fw.jsp 56
Beispiel 2-53 requestbean_inc.jsp 57
Beispiel 2-54 pagebean_param.jsp 58
Beispiel 2-55 *Auszug aus Beispiel* pagebean_param_wert.jsp 58
Beispiel 2-56 pagebean_param_wild.jsp 59
Beispiel 2-57 input_view.jsp 60
Beispiel 2-58 controller.jsp 60
Beispiel 2-59 output_view.jsp 60
Beispiel 2-60 input_view.jsp 68
Beispiel 2-61 controller.jsp 68
Beispiel 2-62 output_view.jsp 69
Beispiel 4-1 web.xml 77
Beispiel 4-2 *Ausschnitt aus* web.xml 78
Beispiel 4-3 *Ausschnitt aus* FunctionWrapper.tld 79
Beispiel 4-4 *Ausschnitt aus* FunctionWrapper.java 79
Beispiel 4-5 index.jsp 79
Beispiel 4-6 *Controller-JSP* aufgabe.jsp 80
Beispiel 4-7 *Ausschnitt aus* aufgabenstellung.jsp 82
Beispiel 4-8 *Ausschnitt aus* korrektur.jsp 83
Beispiel 4-9 *Ausschnitt aus* richtig.jsp 84
Beispiel 4-10 *Ausschnitt aus* ende.jsp 85
Beispiel 4-11 *Servlet-Deklaration in* web.xml 87
Beispiel 4-12 *Taglib-Deklaration* web.xml 87
Beispiel 4-13 *Form-Beans-Deklaration in* struts-config.xml 88
Beispiel 4-14 *Global-Forwards-Definition in* struts-config.xml 88
Beispiel 4-15 *Actionmappings-Defnition in* struts-config.xml 89
Beispiel 4-16 *Actionmappings-Defnition in* struts-config.xml 89
Beispiel 4-17 index.html 89
Beispiel 4-18 *Ausschnitt aus* WiederholenFormBean.jsp 90
Beispiel 4-19 *Ausschnitt aus der Action* Aufgabe.java 92
Beispiel 4-20 aufgabenstellung.jsp, *gekürzt* 95
Beispiel 4-21 *Ausschnitt aus* AntwortFormBean.jsp 96
Beispiel 4-22 *Ausschnitt aus derAction* korrektur.java 96
Beispiel 4-23 *Ausschnitt aus* richtig.jsp 97
Beispiel 4-24 *Ausschnitt aus* ende.jsp 99
Beispiel 4-25 *Ausschnitt aus* web.xml 103
Beispiel 4-26 *Navigationsregeln in* faces-config.xml 104
Beispiel 4-27 *Ausschnitt aus* faces-config.xml 105
Beispiel 4-28 *Ausschnitt aus* index.html 106
Beispiel 4-29 *Ausschnitt aus* index.jsp 107

Beispiel 4-30 *Ausschnitt aus* `web.xml` 107
Beispiel 4-31 *Ausschnitt aus* `aufgabenstellung.jsp` 110
Beispiel 4-32 *von JSF generierter HTML-Code aus* `aufgabenstellung.jsp` 111
Beispiel 4-33 *Ausschnitt aus* `richtig.jsp` 112
Beispiel 4-34 *Ausschnitt aus* `ende.jsp` 114
Beispiel 5-1 *Ausschnitt aus Controller-JSP* `korrektur.jsp` 117
Beispiel 5-2 *Ausschnitt aus der JSF-Seite* `aufgabenstellung.jsp` 117
Beispiel 5-3 *Navigationsregeln in* `faces-config.xml` 118
Beispiel 8-1 *Session-Controller-Servlet* `sessioncontroller.java` 133
Beispiel 8-2 *Session-View-Servlet* `sessionview.java` 133
Beispiel 8-3 *Eingabeseite* `index.html` 133
Beispiel 8-4 *HTML-Header-Datei* `viewheader.html` 133
Beispiel 8-5 *ausgelagerterHTML-Code* `sessionview.html` 134
Beispiel 8-6 *HTML-Footer-Datei* `viewfooter.html` 134
Beispiel 8-7 *Web-App-Konfigurationsdatei* `web.xml` 134
Beispiel 8-8 `Aufgabe.java` 136
Beispiel 8-9 `Aufgabenkatalog.java` 137
Beispiel 8-10 `Testverlauf.java` 139
Beispiel 8-11 `SCInit.java` 139
Beispiel 8-12 `web.xml` 140
Beispiel 8-13 `FunctionWrapper.tld` 140
Beispiel 8-14 `FunctionWrapper.java` 140
Beispiel 8-15 `index.jsp` 140
Beispiel 8-16 `aufgabe.jsp` 141
Beispiel 8-17 `aufgabenstellung.jsp` 142
Beispiel 8-18 `korrektur.jsp` 142
Beispiel 8-19 `richtig.jsp` 142
Beispiel 8-20 `falsch.jsp` 143
Beispiel 8-21 `ende.jsp` 143
Beispiel 8-22 `web.xml` 144
Beispiel 8-23 `struts-config.xml` 144
Beispiel 8-24 `index.html` 144
Beispiel 8-25 `WiederholenFormBean.jsp` 145
Beispiel 8-26 `Aufgabe.java` 145
Beispiel 8-27 `aufgabenstellung.jsp` 147
Beispiel 8-28 `AntwortFormBean.jsp` 147
Beispiel 8-29 `korrektur.java` 147
Beispiel 8-30 `richtig.jsp` 148
Beispiel 8-31 `falsch.jsp` 148
Beispiel 8-32 `ende.jsp` 148
Beispiel 8-33 `web.xml` 149
Beispiel 8-34 `faces-config.xml` 149
Beispiel 8-35 *Ausschnitt aus* `index.html` 149
Beispiel 8-36 `index.jsp` 149
Beispiel 8-37 `aufgabenstellung.jsp` 150
Beispiel 8-38 `richtig.jsp` 151
Beispiel 8-39 `falsch.jsp` 151
Beispiel 8-40 `ende.jsp` 152

Beispiel 8-41 *Beispiel für eine Fragendatei* 153

Literaturverzeichnis

---

[1] M Johann. Alleskönner - Server Erweiterungen mit Java. PC Magazin Juli 1997. Seite 270
[2] M Johann. Kraftzwerge - Server Erweiterungen mit ServletBeans und JHTML. PC Magazin April 1997. Seite 302
[3] J Heid. Hand angelegt - Serverseitiges Scripting mit JSP. iX 11/1998. Seite 68
[4] J Heid. Vieles neu - JSP-Spezifikation 1.0 fertiggestellt. iX 8/1999. Seite 114
[5] JD Davidson, D Coward, Java Servlet Specification Version 2.2, Seite 43 http://java.sun.com/products/servlet/archive.html
[6] P Roßbach .Web-Catering - Tomcat 3.2.1: Servlet- und JSP-Referenzimplementierung. iX 2/2001. Seite 48
[7] L Röwekamp Java Servlet API 2.3 und JavaServer Pages 1.2 - Dynamisch in die nächste Runde Javamagazin 11.2000. Seite 68
[8] JavaServer Pages Specification 0.92, http://www.kirkdorffer.com/jspspecs/jsp092.html
[9] Designing Enterprise Applications with the J2EE Platform, Second Edition http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html
[10] Govind Seshadri, Understanding JavaServer Pages Model 2 architecture http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc_p.html
[11] TMH Reenskaug, The original MVC, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html
[12] http://struts.apache.org/
[13] V Reichel Struts your Web! Javamagazin. 1 2001. p 38
[14] http://jcp.org/aboutJava/communityprocess/final/jsr127/
[15] http://java.sun.com/javaee/javaserverfaces/
[16] R Klute. Mit neuem Gesicht - Java Server Faces vereinfachen Webentwicklung. iX 11/2004. Seite 128
[17] http://myfaces.apache.org/
[18] M Marinschek, A Schnabl. MyFaces: JSF à la Apache. Javamagazin 10.2005. Seite 40
[19] R Meissner. Java serviert - Suns Java-Server bringt Java auf die Serverseite. c't 3/1998. Seite 144.
[20] P Roßbach, H Schreiber. Servlet-Spezifikation: Version 2.1. iX 2/1999. Seite 123
[21] J Hunter, What's new in Java Servlet API 2.2?, http://www.javaworld.com/jw-10-1999/jw-10-servletapi.html
[22] J Heid. Was es sein darf -Servlets als CGI-Alternative: Tools im Überblick. iX 11/1998. Seite 64
[23] http://java.apache.org/jserv/zones.html

[24] J Plachy, J Schmidt. Dynamischer Service – Java-Servlets erzeugen dynamische Web-Inhalte. c't 2/2000. Seite 198
[25] DR Callaway. Inside Servlets. 1st edition. Addison Wesley 1999, Seite 221
[26] V Turau, R Pfeiffer. Java Server Pages. 1. Auflage. Dpunkt.verlag. 2000. Seiten 197-200
[27] DR Callaway. Inside Servlets. 1st edition. Addison Wesley 1999, Seite 254-264
[28] D Reilly. How Do I Use Servlets for State and Session Management?. Dr.Dobb's Journal, May 2000. Seite 111
[29] L Röwekamp, P Rossbach. Quattro Stagioni - JSP-Tutorial, Teil 1: Grundlagen der JavaServer Pages. iX 7/2000. Seite 152
[30] L Röwekamp, P Rossbach. Tonno e cipolla - JSP-Tutorial, Teil 2: Model-View-Controller und Datenbankintegration. iX 8/2000. Seite 148
[31] L Röwekamp, P Rossbach. Frutti di Mare - JSP-Tutorial, Teil 3: TagLibs, XML und Mail. iX 9/2000. Seite 172
[32] V Turau, R Pfeiffer. Java Server Pages. 1. Auflage. Dpunkt.verlag. 2000. Seiten 137-171
[33] http://cocoon.apache.org/
[34] D Wang. Web-Frameworks im Überblick. Javamagazin 9.2004. Seite 36
[35] S Mintert, R Menge. XML verpuppt – Aufbereitung mit Cocoon und Extensible Server Pages. c't 10/2000. Seite 222
[36] M Kolb, A JSTL primer, Part 1-4, http://www.ibm.com/developerworks/java/library/j-jstl0211.html
[37] JavaServer Pages Standard Tag Library, http://www.jcp.org/aboutJava/communityprocess/final/jsr052/
[38] M May, D Marques, R Huß. Zeitsäulen. Java Magazin 1.2003. Seite 77
[39] S Haiges Jakarta Struts. Java Magazin 10.2003. Seite 23
[40] S Haiges, M May. Modular, View, Controller. Javamagazin 10.2003. Seite 28
[41] http://jcp.org/en/jsr/detail?id=168
[42] C Kerpen. Mach auf das Tor – JSR 168, die Portle Spezifikation. Javamagazin 10.2004, Seite 34
[43] C Ziegeler M, Langham. Cocoons neues Portal Framework unterstützt den JSR 168-Portlet-Standard. Javamagazin 10.2004, Seite 43
[44] M Özrürk, M Michel. Einführung in die Konzepte von JSF - GUI furs Web. Javamagazin 1.2004. Seite 38
[45] K Violka, M Weßendorf. Web-Mechanik - Java Server Faces generieren Bedienoberflächen fürs Web. c't 10/2005. Seite 202
[46] D Bartetzko, A Hülsebus, L Röwekamp. Im Angesicht des Web – JSF Tutorial I. iX 4/2006. Seite 136
[47] D Bartetzko, A Hülsebus, L Röwekamp, K Auel. Wie im wirklichen Leben – JSF Tutorial II. iX 5/2006. Seite 154.
[48] D Bartetzko, A Hülsebus, L Röwekamp, U Schnurpfeil, K Auel. Sauber getrennt ist halb gewonnen –JSF Tutorial III. iX 6/2006. Seite 156.
[49] H Braun. Ajax zu Fuß. c't 5/2006. Seite 152.
[50] H Braun. Instant-Ajax. c't 5/2006, Seite 160.
[51] D Wang. Ajax: die alte neue Technologie für Webapplikationen der nächsten Generation. Javamagazin 11.2005. Seite 92.
[52] M Wessendorf. JavaServer Faces Integrationsstrategien für AJAX. 12.2005. Seite 75
[53] D Gruntz, HP Oser. Zieleinlauf mit Java. iX 10/2005. Seite 124.
[54] A Cioroianu, Call JavaBean methods from JSP 2.0 pages, http://www.javaworld.com/javaworld/jw-05-2003/jw-0523-calltag.html
[55] PG Taboada. JSF vs Struts: Weiter geht es mit JavaServer Faces. Javamagazin 10.2005. Seite 28

[56] http://shale.apache.org/
[57] http://www.egjug.org/book/print/73
[58] http://www.barrierefreies-webdesign.de/bitv/bedingungen.php
[59] W Chisholm, G Vanderheiden, Ian Jacobs, W3C - Web Content Accessibility Guidelines 1.0, http://www.w3.org/TR/WAI-WEBCONTENT/

8. Anhang

Anhang 1 Methodenübersicht der Model-Beans

Aufgabe

`public` **`Aufgabe`**`()`
Konstruktor ohne Funktion da eine Bean einen parameterlosen Konstruktor braucht.

```
public Aufgabe(int nr,
               java.lang.String modus,
               java.lang.String fragenintro,
               java.lang.String loesung,
               boolean fragestellen,
               java.lang.String wahlantwort1,
               java.lang.String wahlantwort2,
               java.lang.String wahlantwort3,
               java.lang.String wahlantwort4,
               java.lang.String wahlantwort5,
               java.lang.String bildname,
               java.lang.String hinweis)
```
Konstruktor mit Parameteruebergabe zur Initalisierung der Bean. Die Aufgabe-Bean wird mit allen noetigen Informationen auf einmal gefuettert.
Parameters:
`nr` - Aufgabennummer (muss eindeutig sein)
`modus` - Aufgabenmodus (1=Einfachauswahl, 2=Antwortkombinationsaufgabe, 3=Zuordnungsaufgabe)
`fragenintro` - Fragestellung bzw. Einfuehrung in die Frage
`fragestellen` - soll die Aufgabe gestellt werden? (0=nein, 1=ja)
`wahlantwort1` - Antwortmoeglichkeit 1 bzw A
`wahlantwort2` - Antwortmoeglichkeit 2 bzw B
`wahlantwort3` - Antwortmoeglichkeit 3 bzw C
`wahlantwort4` - Antwortmoeglichkeit 4 bzw D
`wahlantwort5` - Antwortmoeglichkeit 5 bzw E
`bildname` - Name der Bilddatei
`hinweis` - Aufloesung bzw. Hinweis zum finden der Loesung

`public void` **`kopiereProps`**`(Aufgabe aufgabe)`
kopiert alle Eigenschaften vom als Parameter uebergebenen Objekt in das aktuelle Objekt
Parameters:
`aufgabe` - Quell-Objekt

`public void` **`setNr`**`(int nr)`
Setzen der Aufgabennummer
Parameters:
`nr` - Aufgabennummer der aktuellen Aufgabe

`public void` **`setModus`**`(java.lang.String modus)`
Setzen des Aufgabenmodus
Parameters:
`modus` - Aufgabenmodus (1=Einfachauswahl, 2=Antwortkombinationsaufgabe, 3=Zuordnungsaufgabe)

`public void` **`setFragenintro`**`(java.lang.String fragenintro)`
Setzen des Fragenintro-Textes
Parameters:
`fragenintro` - Fragestellung bzw. Fragenintro der aktuellen Aufgabe.

`public void` **`setWahlantwort1`**`(java.lang.String wahlantwort)`
Setzen der Wahlantwort 1
Parameters:
`wahlantwort` - Antwortmoeglichkeit

`public void` **`setWahlantwort2`**`(java.lang.String wahlantwort)`
Setzen der Wahlantwort 2
Parameters:
`wahlantwort` - Antwortmoeglichkeit

`public void` **`setWahlantwort3`**`(java.lang.String wahlantwort)`
Setzen der Wahlantwort 3
Parameters:
`wahlantwort` - Antwortmoeglichkeit

`public void` **`setWahlantwort4`**`(java.lang.String wahlantwort)`
Setzen der Wahlantwort 4
Parameters:
`wahlantwort` - Antwortmoeglichkeit

`public void` **`setWahlantwort5`**`(java.lang.String wahlantwort)`
Setzen der Wahlantwort 5
Parameters:
`wahlantwort` - Antwortmoeglichkeit

`public void` **`setBildname`**`(java.lang.String bildname)`
Setzen des Bildnamens
Parameters:
`bildname` - Dateiname der (externen) Bilddatei

`public void` **`setLoesung`**`(java.lang.String loesung)`
Setzen der Loesung
Parameters:
`loesung` - Loesung der Aufgabenstellung (A-E)

`public void` **`setHinweis`**`(java.lang.String hinweis)`
Setzen des Hinweistextes
Parameters:
`hinweis` - Hinweistext: Aufloesung bzw. Hinweis zum finden der Loesung.

`public void` **`setFragestellen`**`(boolean fragestellen)`
Setzen der Information ob die Frage gestellt werden soll
Parameters:
`fragestellen` - soll die Frage gestellt werden? false=nein, true=ja

`public int` **`getNr`**`()`
Liefert die Aufgabennummer zurück

`public java.lang.String` **`getModus`**`()`
Liefert die Aufgabenmodus zurück

`public java.lang.String` **`getFragenintro`**`()`
Liefert die Fragenintro zurück

`public java.lang.String` **`getWahlantwort1`**`()`
Liefert die Wahlantwort 1 zurück

`public java.lang.String` **`getWahlantwort2`**`()`
Liefert die Wahlantwort 2 zurück

`public java.lang.String` **`getWahlantwort3`**`()`
Liefert die Wahlantwort 3 zurück

`public java.lang.String` **`getWahlantwort4`**`()`
Liefert die Wahlantwort 4 zurück

`public java.lang.String` **`getWahlantwort5`**`()`
Liefert die Wahlantwort 5 zurück

`public java.lang.String` **`getBildname`**`()`
Liefert den Dateinamen des Bildes zurück

`public java.lang.String` **`getLoesung`**`()`
Liefert die richtige Loesung zurück

`public java.lang.String` **`getHinweis`**`()`
Liefert den Hinweistext zurück

`public boolean` **`getFragestellen`**`()`
Liefert zurück ob die Fragen gestellt werden soll (true) oder nicht (false)

Aufgabenkatalog

`public` **`Aufgabenkatalog`**`()`
Konstruktor ohne Funktion da eine Bean einen parameterlosen Konstruktor braucht.

`public void` **`setDateiname`**`(java.lang.String dateiname)`
Setzen des Dateinamens der Datei, welche die Aufgabenliste enthaelt.
Parameters:
`dateiname` - Name der Datei, welche die Aufgabenstellungen als Tab-seperated Values enthaellt.

`public java.util.List` **`getListe`**`()`
Liefert die Aufgabenliste zurueck. Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert. Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.

`public int` **`getAnzahl`**`()`
Liefert die anzahl der Aufgabenstellungen in der Aufgabenliste zurueck. Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert. Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.

`public Aufgabe` **`getAufgabe`**`(int i)`
Liefert die i-te Aufgabe der Aufgabenliste zurueck. Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert. Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.
Parameters:
`i` - Index in der Aufgabe innerhalb der Liste

`public java.lang.Object` **`getReferenz`**`()`
Liefert eine Referenz auf sich selbst (this).

Testverlauf

`public` **`Testverlauf`**`()`
Konstruktor ohne Funktion da eine Bean einen parameterlosen Konstruktor braucht. Startzeit wird gesetzt.

`public void` **`setKatalog`**`(Aufgabenkatalog katalog)`
Setzen der Referenz auf den Katalog bzw. ermitteln der Referenz .
Parameters:
`katalog` - Referenz auf den Aufgabenkatalog oder null

`public void` **`setAntwort`**`(java.lang.String antwort)`
Setzen der vom Benutzer gegebenen Antwort.
Parameters:
`antwort` - Benutzerantwort

`public void` **`setAufgabeWiederholen`**`(boolean wiederholung)`
Aufruf dieser Methode bewirkt dass die aktuelle Aufgabe wiederholt wird. (Sie wird am Ende der Testaufgabenliste angehaengt.)
Parameters:
`wiederholung` - true = wiederholung / false = keine Wiederholung

`public void` **`setTestumfang`**`(int testumfang)`
Setzen des Testumfangs Mittels dieser Methode wird die Anzahl der verschiedenen Fragen festgelegt, die aus dem Aufgabenpool entnommen werden soll.
Parameters:
`testumfang` - Anzahl der verschiedenen Fragen pro Test.

`public Aufgabenkatalog` **`getKatalog`**`()`
Liefert die Referenz auf den Aufgabenkatalog.

`public Aufgabe` **`getAufgabe`**`()`
Liefert die Referenz auf die aktuelle Aufgabe. Sollte noch keine Testaufgabenliste bestehen (=null), so wird eine erzeugt. Wenn keine Referenz zum Aufgabenkatalog besteht (=null), wird null zurueck geliefert.

`public java.lang.String` **`getAntwort`**`()`
Liefert die Benutzerantwort. Es wird die vom Benutzer mittels setAntwort() gegebene Antwort wieder zurueckgeliefert.

`public boolean` **`getAufgabeWiederholen`**`()`
Gibt an ob die Aufgabe wiederholt werden soll. Falls das Ergebnis der Korrektur 'richtig' war, braucht eine Aufgabe nicht wiederholt zu werden = false. Falls das Ergebnis der Korrektur nicht 'richtig' war, soll die Aufgabe erneut gestellt werden = true.

`public int` **`getTestumfang`**`()`
Liefert die Anzahl der verschiedenen zu stellenden Fragen pro Test.

`public int` **`getVerbleibendeAufgaben`**`()`
Liefert die Anzahl der noch zu stellenden Aufgaben.

`public int` **`getGestellteAufgaben`**`()`
Liefert die Anzahl der schon gestellten Aufgaben.

`public int` **`getRichtigeAufgaben`**`()`
Liefert die Anzahl der richtig beantworteten Aufgaben.

`public int` **`getDauer`**`()`
Liefert die Zeit seit Testbeginn. Es wird die Zeit seit Erstellung dieses Objekts (!) = Testbeginn zurueckgeliefert.

`public java.lang.String` **`neueAufgabe`**`()`
'Ereugt' eine neue zu stellende Aufgaben. Aufruf dieser Methode bewirkt das die naechste Aufgabe in der Testliste zur aktuellen Aufgabe wird. Weiters wird der Zaehler der Aufrufe der Methode aufgabekorrigieren() wieder auf 0 gesetzt (damit mehrmaliges Aufrufen der Methode aufgabekorrigieren nicht die Anzahl der richtigen Antworten erhoeht, wie durch ein RELOAD einer Web-Seite vorkommen kann). Gibt es noch eine neue Aufgabe liefert die Methode den String 'neu'. Sind keine Aufgaben mehr vorhanden liefert die Methode den String 'ende'.

`public java.lang.String` **`aufgabekorrigieren`**`()`
Ueberprueft ob die Benutzerantwort gleich der Loesung der aktuellen Aufgabe ist. Jeder Aufruf dieser Methode wird gezaehlt. Nur beim ersten Aufruf wird die Aufgabe als richtig gezaehlt. D.h. mehrmaliges Aufrufen (wie es beim RELOAD einer Web-Seite vorkommen kann) erhoeht nicht die Anzahl der richtigen Antworten. Der Zaehler der Aufrufe wird durch die Methode neueAufgabe() wieder auf 0 gesetzt. Ist die

Aufgabe korrekt beantwortet liefert die Methode den String 'richtig'. Andernfals liefert die Methode den String 'falsch'.

Anhang 2 Vollständige Programmcodes der Beispiel-Web-Anwendungen

Session-Beispiel

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class sessioncontroller extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws ServletException, IOException

  {
    String            wert = null;
    HttpSession       session = null;
    RequestDispatcher rd = null;
    Vector            liste = null;


    wert=request.getParameter("todo");

    session=request.getSession(false);

    if (session==null)
    {

      if ((wert!=null) && (wert.equals("start")))
      {
        session=request.getSession(true);
        rd = request.getRequestDispatcher("sessionview");
        rd.forward (request,response);
        return;
      }
      else
      {
        rd = request.getRequestDispatcher("index.html");
        rd.forward (request,response);
        return;
      }
    }

    if ( (wert!=null) && (wert.equals("ende")) )
    {
      session.invalidate();
      rd = request.getRequestDispatcher("index.html");
      rd.forward (request,response);
      return;
    }

    wert=request.getParameter("zumkorb");

    if (wert!=null)
    {
      liste= (Vector) session.getAttribute("Korb");

      if (liste==null)
      {
        liste = new Vector();
      }
      liste.add(wert);

      session.setAttribute("Korb",liste);
    }

    rd = request.getRequestDispatcher("sessionview");
    rd.forward (request,response);

  }
}
```

Beispiel 8-1
Session-Controller-Servlet `sessioncontroller.java`

```
package at.ac.akhwien.mvc.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
```

```
public class sessionview extends HttpServlet
{

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws ServletException, IOException
  {
    String            wert = null;
    HttpSession       session = null;
    RequestDispatcher rd = null;
    Vector            liste = null;

    session = request.getSession(false);
    if (session==null)
    {
      rd = request.getRequestDispatcher("index.html");
      rd.forward (request,response);
      return;
    }

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    rd = request.getRequestDispatcher("viewheader.html");
    rd.include (request,response);

    rd = request.getRequestDispatcher("sessionview.html");
    rd.include (request,response);

    out.println("Im Korb befinden sich:\n <ol> ");

    liste= (Vector) session.getAttribute("Korb");
    if (liste!=null)
    {
      for (int i=0;i<liste.size();i++)
      out.println("<li>"+liste.get(i)+"</li>\n");
    }
    out.println("</ol> ");

    rd = request.getRequestDispatcher("viewfooter.html");
    rd.include (request,response);

  }
}
```

Beispiel 8-2
Session-View-Servlet `sessionview.java`

```
<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <p>
      <a href="sessioncontroller?todo=start">Start</a>
    </p>
    <p>
      <a href="sessioncontroller?todo=ende">Ende</a>
    </p>
    <form method="GET" action="sessioncontroller"
     <input type="text" name="zumkorb" />
     <input type="submit" value="zum Korb" />
    </form>
  </body>
</html>
```

Beispiel 8-3
Eingabeseite `index.html`

```
<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
```

Beispiel 8-4
HTML-Header-Datei `viewheader.html`

```
    <p>
      <a href="sessioncontroller?todo=start">Start</a>
    </p>
    <p>
      <a href="sessioncontroller?todo=ende">Ende</a>
    </p>
    <form method="GET" action="sessioncontroller"
     <input type="text" name="zumkorb" />
```

```
    <input type="submit" value="zum Korb" />
   </form>
   <hr />
```

Beispiel 8-5
ausgelagerterHTML-Code `sessionview.html`

```
  </body>
</html>
```

Beispiel 8-6
HTML-Footer-Datei `viewfooter.html`

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd" version="2.4">


  <display-name>Session-Demo</display-name>
  <description>
    Diese Webapplication dient zur Illustration von Sessions
  </description>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>session controller</servlet-name>
    <servlet-class>at.ac.akhwien.mvc.servlets.sessioncontroller</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>session view</servlet-name>
    <servlet-class>at.ac.akhwien.mvc.servlets.sessionview</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>session controller</servlet-name>
    <url-pattern>/sessioncontroller</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>session view</servlet-name>
    <url-pattern>/sessionview</url-pattern>
  </servlet-mapping>

</web-app>
```

Beispiel 8-7
Web-App-Konfigurationsdatei `web.xml`

Model-Beans inkl.ServletContext-Listener SCInit

```
package at.ac.akhwien.mvc.model;

/**
 * Diese Bean dient zum Speichern der Daten einer MC-Aufgabenstellung
 * Eine Aufgabenstellung beinhaltet die Fragestellung, Wahlantwortmoeglichkeiten,
 * die richtige Loesung, die vom Pruefungskandidaten eingegebene Antwort sowie
 * eine Referenz zu einer optionalen Bilddatei und einen Hinweistext.
 */


public class Aufgabe
{
  private int      nr;
  private String   modus;
  private String   fragenintro;
  private String   wahlantwort1;
  private String   wahlantwort2;
  private String   wahlantwort3;
  private String   wahlantwort4;
  private String   wahlantwort5;
  private String   bildname;
  private String   loesung;
  private String   hinweis;
  private boolean  fragestellen;

// Construktor

/**
 * Konstruktor ohne Funktion
 * da eine Bean einen parameterlosen Konstruktor braucht.
 */
  public Aufgabe()
  {
  }

/**
 * Konstruktor mit Parameteruebergabe zur Initalisierung der Bean.
 * Die Aufgabe-Bean wird mit allen noetigen Informationen auf einmal gefuettert.
 *
 * @param  nr            Aufgabennummer (muss eindeutig sein)
 * @param  modus         Aufgabenmodus (1-Einfachauswahl, 2-Antwortkombinationsaufgabe, 3-Zuordnungsaufgabe)
 * @param  fragenintro   Fragestellung bzw. Einfuehrung in die Frage
 * @param  fragestellen  soll die Aufgabe gestellt werden? (0-nein, 1-ja)
 * @param  wahlantwort1  Antwortmoeglichkeit 1 bzw A
 * @param  wahlantwort2  Antwortmoeglichkeit 2 bzw B
 * @param  wahlantwort3  Antwortmoeglichkeit 3 bzw C
 * @param  wahlantwort4  Antwortmoeglichkeit 4 bzw D
 * @param  wahlantwort5  Antwortmoeglichkeit 5 bzw E
 * @param  bildname      Name der Bilddatei
 * @param  hinweis       Aufloesung bzw. Hinweis zum finden der Loesung
 */
  public Aufgabe( int      nr,
                  String   modus,
                  String   fragenintro,
                  String   loesung,
                  boolean  fragestellen,
                  String   wahlantwort1,
                  String   wahlantwort2,
                  String   wahlantwort3,
                  String   wahlantwort4,
                  String   wahlantwort5,
                  String   bildname,
                  String   hinweis )
  {
    setNr(nr);
    setModus(modus);
    setFragenintro(fragenintro);
    setLoesung(loesung);
    setFragestellen(fragestellen);
    setWahlantwort1(wahlantwort1);
    setWahlantwort2(wahlantwort2);
    setWahlantwort3(wahlantwort3);
    setWahlantwort4(wahlantwort4);
    setWahlantwort5(wahlantwort5);
    setBildname(bildname);
    setHinweis(hinweis);
  }

/**
 * kopiert alle Eigenschaften
 * vom als Parameter uebergebenen Objekt in das aktuelle Objekt
 *
 * @param  aufgabe Quell-Objekt
 */
  public void kopiereProps(Aufgabe aufgabe)
  {
    setNr(aufgabe.getNr());
    setModus(aufgabe.getModus());
    setFragenintro(aufgabe.getFragenintro());
    setLoesung(aufgabe.getLoesung());
    setFragestellen(aufgabe.getFragestellen());
    setWahlantwort1(aufgabe.getWahlantwort1());
    setWahlantwort2(aufgabe.getWahlantwort2());
    setWahlantwort3(aufgabe.getWahlantwort3());
    setWahlantwort4(aufgabe.getWahlantwort4());
    setWahlantwort5(aufgabe.getWahlantwort5());
    setBildname(aufgabe.getBildname());
    setHinweis(aufgabe.getHinweis());
  }


// set Methoden
// ------------


/**
 * Setzen der Aufgabennummer
 *
 * @param  nr   Aufgabennummer der aktuellen Aufgabe
 */
  public void setNr(int nr)
  {
    this.nr=nr;
  }


/**
 * Setzen des Aufgabenmodus
 *
 * @param  modus   Aufgabenmodus (1-Einfachauswahl, 2-Antwortkombinationsaufgabe, 3-Zuordnungsaufgabe)
 */
  public void setModus(String modus)
  {
    this.modus=modus;
  }


/**
 * Setzen des Fragenintro-Textes
 *
 * @param  fragenintro   Fragestellung bzw. Fragenintro der aktuellen Aufgabe.
 */
  public void setFragenintro(String fragenintro)
  {
    this.fragenintro=fragenintro;
  }


/**
 * Setzen der Wahlantwort 1
 *
 * @param  wahlantwort Antwortmoeglichkeit
 */
  public void setWahlantwort1(String wahlantwort)
  {
    this.wahlantwort1=wahlantwort;
  }


/**
 * Setzen der Wahlantwort 2
 *
 * @param  wahlantwort Antwortmoeglichkeit
 */
  public void setWahlantwort2(String wahlantwort)
  {
    this.wahlantwort2=wahlantwort;
  }


/**
 * Setzen der Wahlantwort 3
 *
 * @param  wahlantwort Antwortmoeglichkeit
 */
  public void setWahlantwort3(String wahlantwort)
  {
    this.wahlantwort3=wahlantwort;
  }


/**
 * Setzen der Wahlantwort 4
 *
 * @param  wahlantwort Antwortmoeglichkeit
 */
  public void setWahlantwort4(String wahlantwort)
  {
    this.wahlantwort4=wahlantwort;
  }


/**
 * Setzen der Wahlantwort 5
 *
 * @param  wahlantwort Antwortmoeglichkeit
 */
  public void setWahlantwort5(String wahlantwort)
  {
    this.wahlantwort5=wahlantwort;
  }
```

```
/**
 * Setzen des Bildnamens
 *
 * @param  bildname Dateiname der (externen) Bilddatei
 */
  public void setBildname(String bildname)
  {
    this.bildname=bildname;
  }


/**
 * Setzen der Loesung
 *
 * @param  loesung Loesung der Aufgabenstellung (A-E)
 */
  public void setLoesung(String loesung)
  {
    this.loesung=loesung;
  }


/**
 * Setzen des Hinweistextes
 *
 * @param  hinweis Hinweistext: Aufloesung bzw. Hinweis zum finden der Loesung.
 */
  public void setHinweis(String hinweis)
  {
    this.hinweis=hinweis;
  }


/**
 * Setzen der Information ob die Frage gestellt werden soll
 *
 * @param  fragestellen soll die Frage gestellt werden? false=nein, true=ja
 */
  public void setFragestellen(boolean fragestellen)
  {
    this.fragestellen=fragestellen;
  }


// get Methoden
// ------------


/**
 * Liefert die Aufgabennummer zurück
 *
 */
  public int getNr()
  {
    return nr;
  }


/**
 * Liefert die Aufgabenmodus zurück
 *
 */
  public String getModus()
  {
    return modus;
  }

/**
 * Liefert die Fragenintro zurück
 *
 */
  public String getFragenintro()
  {
    return fragenintro;
  }


/**
 * Liefert die Wahlantwort 1 zurück
 *
 */
  public String getWahlantwort1()
  {
    return wahlantwort1;
  }


/**
 * Liefert die Wahlantwort 2 zurück
 *
 */
  public String getWahlantwort2()
  {
    return wahlantwort2;
  }

/**
 * Liefert die Wahlantwort 3 zurück
 *
 */
  public String getWahlantwort3()
  {
    return wahlantwort3;
  }


/**
 * Liefert die Wahlantwort 4 zurück
 *
 */
  public String getWahlantwort4()
  {
    return wahlantwort4;
  }


/**
 * Liefert die Wahlantwort 5 zurück
 *
 */
  public String getWahlantwort5()
  {
    return wahlantwort5;
  }


/**
 * Liefert den Dateinamen des Bildes zurück
 *
 */
  public String getBildname()
  {
    return bildname;
  }


/**
 * Liefert die richtige Loesung zurück
 *
 */
  public String getLoesung()
  {
    return loesung;
  }


/**
 * Liefert den Hinweistext zurück
 *
 */
  public String getHinweis()
  {
    return hinweis;
  }


/**
 * Liefert zurück ob die Fragen gestellt werden soll (true) oder nicht (false)
 *
 */
  public boolean getFragestellen()
  {
    return fragestellen;
  }

}
```

Beispiel 8-8
Aufgabe.java

```
package at.ac.akhwien.mvc.model;

import java.util.*;
import java.io.*;


/**
 * Diese Bean laedt die Aufgabenstellungen aus einer Datei in eine Liste.
 */


public class Aufgabenkatalog
{
  private Vector aufgabenliste = null;
  private Aufgabe aufgabe;
  private String message;
  private String dateiname;
```

```
// Konstruktor(en)
// ---------------


/**
 * Konstruktor ohne Funktion
 * da eine Bean einen parameterlosen Konstruktor braucht.
 */
  public Aufgabenkatalog()
  {
  }


// set Methoden
// ------------


/**
 * Setzen des Dateinamens der Datei, welche die Aufgabenliste enthaelt.
 *
 * @param  dateiname Name der Datei, welche die Aufgabenstellungen als Tab-seperated Values enthaellt.
 */
  public void setDateiname(String dateiname)
  {
    this.dateiname=dateiname;
  }



// get Methoden
// ------------


/**
 * Liefert die Aufgabenliste zurueck.
 * Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert.
 * Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.
 *
 */
  private List getListe()
  {
    if (aufgabenliste == null)
    {
      dateneinlesen();
    }

    if (aufgabenliste != null)
    {
      return aufgabenliste;
    }
    else
    {
      return null;
    }
  }


/**
 * Liefert die anzahl der Aufgabenstellungen in der Aufgabenliste zurueck.
 * Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert.
 * Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.
 *
 */
  public int getAnzahl()
  {
    if (aufgabenliste == null)
    {
      dateneinlesen();
    }

    if (aufgabenliste != null)
    {
      return aufgabenliste.size();
    }
    else
    {
      return 0;
    }
  }


/**
 * Liefert die i-te Aufgabe der Aufgabenliste zurueck.
 * Diese get-Methode ueberprueft zuerst ob die Aufgabenliste existiert.
 * Wenn nicht (aufgabenliste==null), dann werden die Daten eingelesen.
 *
 * @param  i   Index in der Aufgabe innerhalb der Liste
 */
  public Aufgabe getAufgabe(int i)
  {
    if (aufgabenliste == null)
    {
      dateneinlesen();
    }

    if (aufgabenliste != null)
    {
      if (aufgabenliste.size()>=i)
      {
        return (Aufgabe) aufgabenliste.get(i);
      }
      else
      {
        return null;
      }
    }
    else
    {
      return null;
    }

  }


// allgemeine Methoden
// -------------------


/**
 * Einlesen der Aufgabenstellungen aus einer Datei
 *
 */
  private void dateneinlesen()
  {
    int             i=0;
    String          zeile;
    String          hilfsstring;
    String[]        aufgabenteil;
    StringTokenizer st;
    BufferedReader  in;

    aufgabenliste = new Vector();
    try
    {
      in = new BufferedReader (new InputStreamReader (
                               new FileInputStream(dateiname)) );
      while( (zeile = in.readLine()) != null )
      {
        aufgabenteil = zeile.split("\\t",12);
        if (Integer.parseInt(aufgabenteil[4])!=0)
        {
            aufgabe = new Aufgabe (Integer.parseInt(aufgabenteil[0]),
                                   aufgabenteil[1], aufgabenteil[2],
                                   aufgabenteil[3], aufgabenteil[4].equals("1"),
                                   aufgabenteil[5], aufgabenteil[6],
                                   aufgabenteil[7], aufgabenteil[8],
                                   aufgabenteil[9], aufgabenteil[10],
                                   aufgabenteil[11]);
          aufgabenliste.add(aufgabe);
        }
      }
      in.close();
    }
    catch (Exception e)
    {
      message=e.toString();
      System.out.println("error " + e);
    }
  }

  public Object getReferenz()
  {
    return (this);
  }

}
```

Beispiel 8-9
Aufgabenkatalog.java

```
package at.ac.akhwien.mvc.model;

import java.lang.*;
import java.util.*;

/**
 * Diese Bean enthaelt alle Daten die einen einzelnen Test betreffen.
 * Es wird eine Verbindung zur Aufgabenkatalog-Bean hergestellt und eine
 * Aufgabenliste erstellt die aus Referenzen auf die Aufgaben-Daten im der
 * Aufgabenkatalog-Bean besteht (Daten werden 'durchgereicht').
 * Weiters sind Methoden vorhanden die Antwort zu ueberpruefen und zur
 * naechsten Aufgabe weiterzuleiten.
 */

public class Testverlauf
{
  private Aufgabenkatalog   katalog;
  private int               aktuell;
  private Date              startzeit;
```

```
  private int               testumfang;
  private Vector            testaufgabenliste;
  private String            antwort;
  private int               korregierenaufrufe;
  private int               richtig;
  private String            ergebnis="";


// Konstruktor(en)
// ---------------

/**
 * Konstruktor ohne Funktion
 * da eine Bean einen parameterlosen Konstruktor braucht.
 * Startzeit wird gesetzt.
 */
  public Testverlauf()
  {
    startzeit = new Date();
    testaufgabenliste = null;
  }


// set Methoden
// ------------


/**
 * Setzen der Referenz auf den Katalog bzw ermitteln der Referenz .
 *
 * @param  katalog Referenz auf den Aufgabenkatalog oder null
 */
  public void setKatalog(Aufgabenkatalog katalog)
  {
    this.katalog = katalog;
  }


/**
 * Setzen der vom Benutzer gegebenen Antwort.
 *
 * @param  antwort Benutzerantwort
 */
  public void setAntwort(String antwort)
  {
    this.antwort=antwort;
  }


/**
 * Aufruf dieser Methode bewirkt dass die aktuelle Aufgabe wiederholt wird.
 * (Sie wird am Ende der Testaufgabenliste angehaengt.)
 *
 * @param  wiederholung true = wiederholung  /  false = keine Wiederholung
 */
  public void setAufgabeWiederholen(boolean wiederholung)
  {
    if (wiederholung==true)
    {
      testaufgabenliste.add(testaufgabenliste.get(aktuell));
    }
  }

/**
 * Setzen des Testumfangs
 * Mittels dieser Methode wird die Anzahl der verschiedenen Fragen festgelegt,
 * die aus dem Aufgabenpool entnommen werden soll.
 *
 * @param  testumfang Anzahl der verschiedenen Fragen pro Test.
 */
  public void setTestumfang(int testumfang)
  {
    if (testumfang < 0)
    {
      testumfang = 0;
    }
    this.testumfang=testumfang;
  }



// get Methoden
// ------------


/**
 * Liefert die Referenz auf den Aufgabenkatalog.
 *
 */
  public Aufgabenkatalog getKatalog()
  {
    return(katalog);
  }


/**
 * Liefert die Referenz auf die aktuelle Aufgabe.
 *
 * Sollte noch keine Testaufgabenliste bestehen (=null), so wird eine erzeugt.
 * Wenn keine Referenz zum Aufgabenkatalog besteht (=null), wird null zurueck
 * geliefert.
 *
 */
  public Aufgabe getAufgabe()
  {
    int zufall;
    int katalogumfang;
    Aufgabe aufgabe;

    katalog = getKatalog();
    if (katalog == null)
    {
      return(null);
    }
    else
    {
      if (testaufgabenliste==null)
      {
        testaufgabenliste = new Vector();

        katalogumfang=katalog.getAnzahl();
        if (testumfang>katalogumfang)
          { testumfang = katalogumfang; }

        for ( int i =0; i<testumfang; i=i+1 )
        {
          do
          {
            zufall = (int) Math.floor ( (double) katalogumfang*Math.random());
            aufgabe=katalog.getAufgabe(zufall);
          } while (testaufgabenliste.contains(aufgabe));
          testaufgabenliste.add(aufgabe);
        }
        aktuell = 0;
      }
      return ((Aufgabe) testaufgabenliste.get(aktuell));
    }
  }



/**
 * Liefert die Benutzerantwort.
 *
 * Es wird die vom Benutzer mittels setAntwort() gegebene Antwort wieder
 * zurueckgeliefert.
 *
 */
  public String getAntwort()
  {
    return this.antwort;
  }



/**
 * Gibt an ob die Aufgabe wiederholt werden soll.
 *
 * Falls das Ergebnis der Korrektur 'richtig' war, braucht eine Aufgabe nicht
 * wiederholt zu werden = false.
 * Falls das Ergebnis der Korrektur nicht 'richtig' war, soll die Aufgabe erneut
 * gestellt werden = true.
 *
 */
  public boolean getAufgabeWiederholen()
  {
    if (ergebnis.equals("richtig"))
    {
      return(false);
    }
    else
    {
      return(true);
    }
  }



/**
 * Liefert die Anzahl der verschiedenen zu stellenden Fragen pro Test.
 *
 */
  public int getTestumfang()
  {
    return (testumfang);
  }



/**
 * Liefert die Anzahl der noch zu stellenden Aufgaben.
 *
 */
  public int getVerbleibendeAufgaben()
  {
    return (testaufgabenliste.size()-aktuell-1);
  }


/**
```

```
 * Liefert die Anzahl der schon gestellten Aufgaben.
 *
 */
  public int getGestellteAufgaben()
  {
    return (aktuell+1);
  }

/**
 * Liefert die Anzahl der richtig beantworteten Aufgaben.
 *
 */
  public int getRichtigeAufgaben()
  {
    return (richtig);
  }

/**
 * Liefert die Zeit seit Testbeginn.
 *
 * Es wird die Zeit seit Erstellung dieses Objekts (!) = Testbeginn
 * zurueckgeliefert.
 *
 */
  public int getDauer()
  {
    Date   jetzt = new Date();
    return ((int) (jetzt.getTime() - startzeit.getTime())/1000);
  }



// allgemeine oeffentliche Methoden
// --------------------------------


/**
 * 'Ereugt' eine neue zu stellende Aufgaben.
 *
 * Aufruf dieser Methode bewirkt das die naechste Aufgabe in der Testliste zur
 * aktuellen Aufgabe wird.
 * Weiters wird der Zaehler der Aufrufe der Methode aufgabekorrigieren() wieder
 * auf 0 gesetzt (damit mehrmaliges Aufrufen der Methode aufgabekorrigieren
 * nicht die Anzahl der richtigen Antworten erhoeht, wie durch ein RELOAD einer
 * Web-Seite vorkommen kann).
 * Gibt es noch eine neue Aufgabe liefert die Methode den String 'neu'.
 * Sind keine Aufgaben mehr vorhanden liefert die Methode den String 'ende'.
 *
 */
  public String neueAufgabe()
  {
    // Ist die Liste der Testaufgaben noch gar nicht erstellt,
    // da noch keine Aufruf von getAufgabe() erfolgt war (= Ist dies der
    // erste Aufruf von neueAufgabe()?)
    if (testaufgabenliste==null)
    {
      return ("neu");
    }
    else
    {
      if(getVerbleibendeAufgaben()>0)
      {
        // Auf naechste Aufgabe zeigen
        aktuell = aktuell +1;

        // Der Zaehler der Aufrufe der Methode aufgabekorrigieren()
        // wird wieder auf 0 gesetzt.
        korregierenaufrufe=0;
        return ("neu");
      }
      else
      {
        return ("ende");
      }
    }
  }

/**
 * Ueberprueft ob die Benutzerantwort gleich der Loesung der aktuellen Aufgabe
 * ist.
 *
 * Jeder Aufruf dieser Methode wird gezaehlt. Nur beim ersten Aufruf wird die
 * Aufgabe als richtig gezaehlt. D.h. mehrmaliges Aufrufen (wie es beim RELOAD
 * einer Web-Seite vorkommen kann) erhoeht nicht die Anzahl der richtigen
 * Antworten. Der Zaehler der Aufrufe wird durch die Methode neueAufgabe()
 * wieder auf 0 gesetzt.
 * Ist die Aufgabe korrekt beantwortet liefert die Methode den String 'richtig'.
 * Andernfals liefert die Methode den String 'falsch'.
 *
 */
  public String aufgabekorrigieren()
  {

    if (getAufgabe().getLoesung().equals(getAntwort()))
    {
      korregierenaufrufe = korregierenaufrufe +1;

      // Nur beim ersten Aufruf wird die Aufgabe als richtig gezaehlt.
      if (korregierenaufrufe==1)
      {
        richtig = richtig +1;
      }
      ergebnis = "richtig";
      return ("richtig");
    }
    else
    {
      ergebnis="falsch";
      return ("falsch");
    }
  }

}
```

Beispiel 8-10
Testverlauf.java

```
package at.ac.akhwien.mvc.model;

import java.util.*;
import java.io.*;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

/**
 * Dieser Listener sorgt dafuer, dass der Aufgabenkatalog von den
 * Session-Beans gefunden werden kann.
 * Der Listener ermittelt den Ort der Datenbank-Datei innerhalb des
 * Dateisystems und legt diesen im application-Context der Web-Applikation ab.
 */

public class SCInit implements ServletContextListener {

  public void contextInitialized(ServletContextEvent sce)
  {
    ServletContext       application;
    String               rel_dateiname;

    application = sce.getServletContext();
    rel_dateiname=(String) application.getInitParameter("rel_dateiname");

    application.setAttribute("dateiname",
                             application.getRealPath(rel_dateiname));
  }

  public void contextDestroyed(ServletContextEvent sce)
  {

  }

}
```

Beispiel 8-11
SCInit.java

JSTL-Version der Knoko-Lernprogramm-Webapplikation

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd" version="2.4">


  <display-name>Knoko Pruefungsvorbereitung</display-name>
  <description>
    Diese Webapplication dient zur Pruefungsvorbereitung fuer das
    Knochenkolloquium.
  </description>


  <context-param>
    <param-name>rel_dateiname</param-name>
    <param-value>/WEB-INF/knokofragen.txt</param-value>
  </context-param>

  <context-param>
    <param-name>testumfang</param-name>
    <param-value>6</param-value>
  </context-param>

  <listener>
    <listener-class>at.ac.akhwien.mvc.model.SCInit</listener-class>
  </listener>

  <taglib>
    <taglib-uri>http://akh-wien.ac.at/functionwrapper</taglib-uri>
    <taglib-location>/WEB-INF/FunctionWrapper.tld </taglib-location>
  </taglib>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Beispiel 8-12
web.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
    version="2.0">

   <tlib-version>1.0</tlib-version>
   <short-name>fw</short-name>
   <uri>http://akh-wien.ac.at/functionwrapper</uri>


   <function>
     <name>neueAufgabe</name>

     <function-class>
       at.ac.akhwien.mvc.taglibs.FunctionWrapper
     </function-class>

     <function-signature>
       java.lang.String neueAufgabe(at.ac.akhwien.mvc.model.Testverlauf)
     </function-signature>

   </function>


   <function>
     <name>aufgabekorrigieren</name>

     <function-class>
       at.ac.akhwien.mvc.taglibs.FunctionWrapper
     </function-class>

     <function-signature>
       java.lang.String aufgabekorrigieren(at.ac.akhwien.mvc.model.Testverlauf)
     </function-signature>

   </function>

</taglib>
```

Beispiel 8-13
FunctionWrapper.tld

```
package at.ac.akhwien.mvc.taglibs;

import at.ac.akhwien.mvc.model.Testverlauf;

/**
 * Diesese Klasse ist nur eine H&uuml;lle (Wrapper) f&uuml; die Model-Beans
 * um EL Funktionsaufrufe entsprechend der JSP 2.0 Spezifikation
 * zu erm&ouml;glichen.
 */

public class FunctionWrapper
{

/**
 * Aufruf der Methode 'neueAufgabe()' aus der Bean Testverlauf
 * Liefert den unver&auml;nderten R&uuml;ckgabewert der Methode neueAufgabe()
 *
 * @param  tvObject   Objektreferenz auf Testverlauf-Objekt
 */
  public static String neueAufgabe(Testverlauf tvObject)
  {
    return tvObject.neueAufgabe();
  }

/**
 * Aufruf der Methode 'aufgabekorrigieren()' aus der Bean Testverlauf
 * Liefert den unver&auml;nderten R&uuml;ckgabewert der Methode
 * aufgabekorrigieren()
 *
 * @param  tvObject   Objektreferenz auf Testverlauf-Objekt
 */
  public static String aufgabekorrigieren(Testverlauf tvObject)
  {
    return tvObject.aufgabekorrigieren();
  }

}
```

Beispiel 8-14
FunctionWrapper.java

```
<%@page  session="false" %>

<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabe.jsp">Start</a>.

    </hr>

  </body>
</html>
```

Beispiel 8-15
index.jsp

```
%@page session="true" %>

<%@ taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fw"   uri="http://akh-wien.ac.at/functionwrapper" %>

<jsp:useBean id="Aufgabenkatalog"
     class="at.ac.akhwien.mvc.model.Aufgabenkatalog" scope="application">
  <jsp:setProperty name="Aufgabenkatalog"
                   property="dateiname" value="${applicationScope.dateiname}" />
</jsp:useBean>

<jsp:useBean id="Testverlauf"
                   class="at.ac.akhwien.mvc.model.Testverlauf" scope="session">
  <jsp:setProperty name="Testverlauf"
                   property="katalog" value="${Aufgabenkatalog.referenz}" />
  <jsp:setProperty name="Testverlauf"
```

```
                    property="testumfang" value="${initParam.testumfang}" />
</jsp:useBean>


<jsp:setProperty name="Testverlauf" property="aufgabeWiederholen" />

<c:set var="neueaufgabe" value="${fw:neueAufgabe(Testverlauf)}" />

<c:choose>
  <c:when test="${ neueaufgabe eq 'neu'}" >
    <jsp:forward page="aufgabenstellung.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="ende.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 8-16
aufgabe.jsp

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf" class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>



<html>
  <head>
    <title>Knoko-Lernprogramm Frage</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    Fragennummer  ${Testverlauf.aufgabe.nr},
    <form method="post" action="korrektur.jsp">

      <c:choose>
        <c:when test="${Testverlauf.aufgabe.modus==1}" >
          Einfachauswahl (A)
          <hr/>

          <table>
            <tr>
              <td style="zeile"> ${Testverlauf.aufgabe.fragenintro} </td>
            </tr>
            <c:choose>
              <c:when test="${Testverlauf.aufgabe.bildname!=''}" >
                <tr>
                  <td > <img src="${Testverlauf.aufgabe.bildname}" /> </td>
                </tr>
              </c:when>
            </c:choose>
          </table>

          <table>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="A"> </td>
                 <td> A </td>
                 <td> ${Testverlauf.aufgabe.wahlantwort1} </td>
              </label>
            </tr>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="B"> </td>
                 <td> B </td>
                 <td> ${Testverlauf.aufgabe.wahlantwort2} </td>
              </label>
            </tr>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="C"> </td>
                 <td> C </td>
                 <td> ${Testverlauf.aufgabe.wahlantwort3} </td>
              </label>
            </tr>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="D"> </td>
                 <td> D </td>
                 <td> ${Testverlauf.aufgabe.wahlantwort4} </td>
              </label>
            </tr>
            <tr>
              <label>
                 <td> <input type="radio" name="antwort" value="E"> </td>
                 <td> E </td>
                 <td> ${Testverlauf.aufgabe.wahlantwort5} </td>
              </label>
            </tr>
          </table><p></p>
        </c:when>

        <c:when test="${Testverlauf.aufgabe.modus==2}" >
          Antwortkombinationsaufgabe (Kprim)
          <hr/>

          <p> ${Testverlauf.aufgabe.fragenintro}
          <table>
            <tr>
              <td> 1 </td> <td> ${Testverlauf.aufgabe.wahlantwort1} </td>
            </tr>
            <tr>
              <td> 2 </td> <td> ${Testverlauf.aufgabe.wahlantwort2} </td>
            </tr>
            <tr>
              <td> 3 </td> <td> ${Testverlauf.aufgabe.wahlantwort3} </td>
            </tr>
          </table><p>
          <table width="70%">
            <tr>
              <td align="center" width="20%">
                <label>
                  A <input type="radio" name="antwort" value="A">
                </label>
              </td>
              <td align="center" width="20%">
                <label>
                  B <input type="radio" name="antwort" value="B">
                </label>
              </td>
              <td align="center" width="20%">
                <label>
                  C <input type="radio" name="antwort" value="C">
                </label>
              </td>
              <td align="center" width="20%">
                <label>
                  D <input type="radio" name="antwort" value="D">
                </label>
              </td>
              <td align="center" width="20%">
                <label>
                  E <input type="radio" name="antwort" value="E">
                </label>
              </td>
            </tr>
            <tr>
              <td align="center">1+2+3</td>
              <td align="center">1+2</td>
              <td align="center">2+3</td>
              <td align="center">1</td>
              <td align="center">3</td>
            </tr>
          </table>
        </c:when>

        <c:when test="${Testverlauf.aufgabe.modus==3}" >
          Zuordnungsfrage (B)
          <hr />

          <table>
            <tr>
              <td style="zeile"> ${Testverlauf.aufgabe.fragenintro} </td>
            </tr>
            <c:choose>
              <c:when test="${Testverlauf.aufgabe.bildname!=''}" >
                <tr>
                  <td > <img src="${Testverlauf.aufgabe.bildname}" /> </td>
                </tr>
                <tr>
                  <td > Wenn der gefragte Begriff durch 'A' - 'D' nicht
                        bezeichnet ist, ist 'E' zu wählen. </td>
                </tr>
              </c:when>
            </c:choose>
          </table>

          <c:choose>
            <c:when test="${Testverlauf.aufgabe.wahlantwort1!=''}" >
              <table>
                <tr>
                  <td> A </td> <td> ${Testverlauf.aufgabe.wahlantwort1} </td>
                </tr>
                <tr>
                  <td> B </td> <td> ${Testverlauf.aufgabe.wahlantwort2} </td>
```

```
              </tr>
              <tr>
                <td> C </td> <td> ${Testverlauf.aufgabe.wahlantwort3} </td>
              </tr>
              <tr>
                <td> D </td> <td> ${Testverlauf.aufgabe.wahlantwort4} </td>
              </tr>
              <tr>
                <td> E </td> <td> ${Testverlauf.aufgabe.wahlantwort5} </td>
              </tr>
            </table><p></p>
          </c:when>
        </c:choose>

        <table width="70%">
          <tr>
            <td align="center" width="20%">
              <label>A <input type="radio" name="antwort" value="A"> </label>
            </td>
            <td align="center" width="20%">
              label>B <input type="radio" name="antwort" value="B"> </label>
            </td>
            <td align="center" width="20%">
              <label>C <input type="radio" name="antwort" value="C"> </label>
            </td>
            <td align="center" width="20%">
               <label>D <input type="radio" name="antwort" value="D"> </label>
            </td>
            <td align="center" width="20%">
              <label>E <input type="radio" name="antwort" value="E"> </label>
            </td>
          </tr>
        </table>

       </c:when>
     </c:choose>

     <input type="submit" value="Antwort abschicken">
   </form>

   <hr />

   Zeit seit Beginn: ${Testverlauf.dauer} Sekunde(n).
   Es wurden bisher ${Testverlauf.gestellteAufgaben} Fragen gestellt,
   davon richtig: ${Testverlauf.richtigeAufgaben}.

   <hr />
   <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>


  </body>
</html>
```

Beispiel 8-17
aufgabenstellung.jsp

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fw"   uri="http://akh-wien.ac.at/functionwrapper" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<jsp:setProperty name="Testverlauf" property="antwort" />

<c:set var="korrektur" value="${fw:aufgabekorrigieren(Testverlauf)}" />

<c:choose>
  <c:when test="${ korrektur eq 'richtig'}" >
    <jsp:forward page="richtig.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="falsch.jsp" />
  </c:otherwise>
</c:choose>
```

Beispiel 8-18
korrektur.jsp

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf" class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <head>
    <title>Richtig</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    Fragennummer  ${Testverlauf.aufgabe.nr}
    <hr />
    <h1>Richtig!</h1>

    <p>Gegebene Antwort: ${Testverlauf.antwort},
       richtige Antwort: ${Testverlauf.aufgabe.loesung} </p>

    <form METHOD="GET" ACTION="aufgabe.jsp">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true" />
        diese Aufgabe nocheinmal stellen
      </p>
      <input type="submit" value="weiter">
    </form>

    <hr />

    Zeit seit Beginn: ${Testverlauf.dauer} Sekunde(n).
    Es wurden bisher ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    davon richtig: ${Testverlauf.richtigeAufgaben}.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 8-19
richtig.jsp

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <head>
    <title>Falsch</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>
```

```
    Fragennummer  ${Testverlauf.aufgabe.nr}
    <hr />
    <h1>Falsch!</h1>

    <p>Gegebene Antwort: ${Testverlauf.antwort},
       richtige Antwort: ${Testverlauf.aufgabe.loesung} </p>

    <form METHOD="GET" ACTION="aufgabe.jsp">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true"
               checked="checked" />
        diese Aufgabe nocheinmal stellen
      </p>
      <input type="submit" value="weiter">
    </form>

    <hr />

    Zeit seit Beginn: ${Testverlauf.dauer} Sekunde(n).
    Es wurden bisher ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    davon richtig: ${Testverlauf.richtigeAufgaben}.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 8-20
falsch.jsp

```
<%@page session="true" %>

<%@taglib prefix="c"    uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>


<jsp:useBean id="Testverlauf" class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <sess:invalidate/>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <head>
    <title>Richtig</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    <h1>Ende</h1>
    <h2>Auswertung:</h2>

    <p>
    Vom Computer wurden ${Testverlauf.testumfang} verschieden Fragen
    aus einem Pool von ${Testverlauf.katalog.anzahl} Aufgaben per Zufall ausgesucht.
    </p>

    <p>
    Die Dauer zur Beantwortung aller Fragen betrug ${Testverlauf.dauer} Sekunde(n).
    Dies entspricht ${Testverlauf.dauer/Testverlauf.gestellteAufgaben} Sekunden pro Frage.
    </p>

    <p>
    Insgesammt wurden ${Testverlauf.gestellteAufgaben} Fragen gestellt,
    von denen Sie ${Testverlauf.richtigeAufgaben} richtig beantwortet haben
    (${ 100*Testverlauf.richtigeAufgaben / Testverlauf.gestellteAufgaben} % )
    </p>


    <hr />
    <a href="index.jsp">zum Start</a>

  </body>
</html>

<sess:invalidate/>
```

Beispiel 8-21
ende.jsp

Struts-Version der Knoko-Lernprogramm-Webapplikation

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">


  <display-name>Knoko Pruefungsvorbereitung</display-name>
  <description>
    Diese Webapplication dient zur Pruefungsvorbereitung fuer das
    Knochenkolloquium.
  </description>

  <!-- Standardmaessiges Action Servlet Konfiguration (mit Debugging) -->
  <servlet>
    <servlet-name>action</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>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>


  <!-- Standardmaessiges Action Servlet Mapping mit *.do -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- Struts Tag Library Descriptoren -->
  <taglib>
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/tags/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/tags/struts-nested</taglib-uri>
    <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/tags/struts-tiles</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
  </taglib>

  <context-param>
    <param-name>rel_dateiname</param-name>
    <param-value>/WEB-INF/knokofragen.txt</param-value>
  </context-param>

  <context-param>
    <param-name>testumfang</param-name>
    <param-value>6</param-value>
  </context-param>

  <listener>
    <listener-class>at.ac.akhwien.mvc.model.SCInit</listener-class>
  </listener>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Beispiel 8-22
web.xml

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

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">


<struts-config>

  <!-- ========== Form Bean Definitions =================================== -->

  <form-beans>
    <form-bean      name="AntwortForm"
                    type="at.ac.akhwien.mvc.formbeans.AntwortFormBean"/>
    <form-bean      name="WiederholenForm"
                    type="at.ac.akhwien.mvc.formbeans.WiederholenFormBean"/>
  </form-beans>

  <!-- ========== Global Forwards ========================================= -->

  <global-forwards>

    <forward name="index" path="/index.jsp" />

  </global-forwards>

  <!-- ========== Action Mapping Definitions ============================== -->

  <action-mappings>

    <action    path="/aufgabe"
               type="at.ac.akhwien.mvc.actions.Aufgabe"
               scope="request"
               name="WiederholenForm" >
      <forward name="neu"  path="/aufgabenstellung.jsp"/>
      <forward name="ende"  path="/ende.jsp"/>
    </action>

    <action    path="/korrektur"
               type="at.ac.akhwien.mvc.actions.Korrektur"
               scope="request"
               name="AntwortForm" >
      <forward name="richtig"  path="/richtig.jsp"/>
      <forward name="falsch"   path="/falsch.jsp"/>
    </action>


  </action-mappings>

  <message-resources parameter="resources.ApplicationResources" />

</struts-config>
```

Beispiel 8-23
struts-config.xml

```
<%@page  session="false" %>

<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabe.do">Start</a>.

    </hr>

  </body>
</html>
```

Beispiel 8-24
index.html

```
package at.ac.akhwien.mvc.formbeans;

import org.apache.struts.action.ActionForm;

public final class WiederholenFormBean extends ActionForm
{
```

```
  // Eigenschaften
  private String aufgabeWiederholen = "";

  // Methoden

  public void setAufgabeWiederholen (String aufgabeWiederholen_neu)
  {
    this.aufgabeWiederholen = aufgabeWiederholen_neu;
  }

  public String getAufgabeWiederholen ()
  {
    return this.aufgabeWiederholen;
  }
}
```

Beispiel 8-25
WiederholenFormBean.jap

```
package at.ac.akhwien.mvc.actions;

import java.io.*;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import org.apache.commons.beanutils.PropertyUtils;

import at.ac.akhwien.mvc.model.Aufgabenkatalog;
import at.ac.akhwien.mvc.model.Testverlauf;


/**
 * Aufgabe
 *
 * @author Harald Kornfeil
 */

public final class Aufgabe extends Action
{
  /**
  * Dient zur Abarbeitung eines HTTP Request.
  * Es wird die entsprechende Response generiert bzw. auf eine andere
  * Web-Komponente weitergeleitet, die sie dann erstellt.
  *
  * Es wird eine <code>ActionForward</code> Instanz zurückgegeben, die angiebt
  * wohin/wie weitergeleitet werden soll (oder <code>null</code> wenn die
  * Antwort bereits generiert wurde.
  *
  * @param mapping die ActionMapping die verwendet wird
  * @param form eine optionale ActionForm-Bean für diesen Request
  * @param request der HTTP request um den es geht
  * @param response die HTTP response die wir erzeugen
  *
  * @exception eine Exception falls Business-Logik eine Exception wirft
  */

  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) throws Exception
  {
    Aufgabenkatalog                 aufgabenkatalog;
    Testverlauf          testverlauf;
    at.ac.akhwien.mvc.model.Aufgabe   aufgabe;
    ServletConfig        config;
    ServletContext       application;
    HttpSession          session;
    String              wiederholen;
    String              weiter;


    config = getServlet().getServletConfig();
    application = config.getServletContext();
    session=request.getSession(false);

    aufgabenkatalog = (at.ac.akhwien.mvc.model.Aufgabenkatalog)
                      application.getAttribute("Aufgabenkatalog");
    if (aufgabenkatalog==null)
    {
      aufgabenkatalog = new Aufgabenkatalog();
      aufgabenkatalog.setDateiname( (java.lang.String)
                                    application.getAttribute("dateiname") );
      application.setAttribute("Aufgabenkatalog",aufgabenkatalog);
    }

    testverlauf= (at.ac.akhwien.mvc.model.Testverlauf)
                 session.getAttribute("Testverlauf");
    if (testverlauf==null)
    {
      testverlauf = new Testverlauf();
      testverlauf.setKatalog((at.ac.akhwien.mvc.model.Aufgabenkatalog)
                             aufgabenkatalog.getReferenz());
      testverlauf.setTestumfang(Integer.parseInt(
                                application.getInitParameter("testumfang")));
      session.setAttribute("Testverlauf",testverlauf);

    }
    else
    {
      wiederholen = (String) PropertyUtils.getSimpleProperty
                                                  (form, "aufgabeWiederholen");
      if (wiederholen.equalsIgnoreCase("true"))
      {
        testverlauf.setAufgabeWiederholen(true);
      }
    }

    weiter=testverlauf.neueAufgabe();

    return (mapping.findForward(weiter));


  }

}
```

Beispiel 8-26
Aufgabe.java

```
<%@page  session="true" %>

<%@ taglib prefix="html"  uri="/tags/struts-html" %>
<%@ taglib prefix="bean"  uri="/tags/struts-bean" %>
<%@ taglib prefix="logic" uri="/tags/struts-logic" %>

<html>
  <head>
    <title>Knoko-Lernprogramm Frage</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>

  <body bgcolor=white>
    Fragennummer  <bean:write name="Testverlauf" property="aufgabe.nr"/>
      <html:form action="/korrektur.do">
        <logic:equal name="Testverlauf" property="aufgabe.modus" value="1">
          Einfachauswahl (A)
          <hr/>

          <table>
            <tr>
              <td style="zeile">
                <bean:write name="Testverlauf"
                            property="aufgabe.fragenintro" filter="false" />
              </td>
            </tr>
            <logic:notEqual name="Testverlauf"
                            property="aufgabe.bildname" value="">
              <tr>
                <td > <img src="${Testverlauf.aufgabe.bildname}" /> </td>
              </tr>
            </logic:notEqual>
          </table>

          <table>
            <tr>
              <label>
                <td> <input type="radio" name="antwort" value="A"> </td>
                <td> A </td>
                <td>
                  <bean:write name="Testverlauf"
                              property="aufgabe.wahlantwort1"/>
                </td>
              </label>
            </tr>
            <tr>
```

```
          <label>
            <td> <input type="radio" name="antwort" value="B"> </td>
            <td> B </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort2"/>
            </td>
          </label>
        </tr>
        <tr>
          <label>
            <td> <input type="radio" name="antwort" value="C"> </td>
            <td> C </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort3"/>
            </td>
          </label>
        </tr>
        <tr>
          <label>
            <td> <input type="radio" name="antwort" value="D"> </td>
            <td> D </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort4"/>
            </td>
          </label>
        </tr>
        <tr>
          <label>
            <td> <input type="radio" name="antwort" value="E"> </td>
            <td> E </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort5"/>
            </td>
          </label>
        </tr>
      </table><p></p>
    </logic:equal>

    <logic:equal name="Testverlauf" property="aufgabe.modus" value="2">
      Antwortkombinationsaufgabe (Kprim)
      <hr/>

      <p> <bean:write name="Testverlauf" property="aufgabe.fragenintro"/>
      <table>
         <tr>
          <td> 1 </td>
          <td>
            <bean:write name="Testverlauf"
                        property="aufgabe.wahlantwort1"/>
          </td>
        </tr>
        <tr>
          <td> 2 </td>
          <td>
            <bean:write name="Testverlauf"
                        property="aufgabe.wahlantwort2"/>
          </td>
        </tr>
        <tr>
          <td> 3 </td>
          <td>
            <bean:write name="Testverlauf"
                        property="aufgabe.wahlantwort3"/>
          </td>
        </tr>
      </table><p>
      <table width="70%">
        <tr>
          <td align="center" width="20%">
            <label>
              A <input type="radio" name="antwort" value="A">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              B <input type="radio" name="antwort" value="B">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              C <input type="radio" name="antwort" value="C">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              D <input type="radio" name="antwort" value="D">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              E <input type="radio" name="antwort" value="E">
            </label>
          </td>
        </tr>
        <tr>
          <td align="center">1+2+3</td>
          <td align="center">1+2</td>
          <td align="center">2+3</td>
          <td align="center">1</td>
          <td align="center">3</td>
        </tr>
      </table>

    </logic:equal>

    <logic:equal name="Testverlauf" property="aufgabe.modus" value="3">
      Zuordnungsfrage (B)
      <hr />

      <table>
        <tr>
          <td style="zeile">
            <bean:write name="Testverlauf" property="aufgabe.fragenintro"/>
          </td>
        </tr>
        <logic:notEqual name="Testverlauf"
                        property="aufgabe.bildname" value="">
          <tr>
            <td > <img src="${Testverlauf.aufgabe.bildname}" /> </td>
          </tr>
          <tr>
            <td >
              Wenn der gefragte Begriff durch 'A' - 'D' nicht bezeichnet
              ist, ist 'E' zu wählen.
            </td>
          </tr>
        </logic:notEqual>
      </table>

      <logic:notEqual name="Testverlauf"
                      property="aufgabe.wahlantwort1" value="">
        <table>
          <tr>
            <td> A </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort1"/>
            </td>
          </tr>
          <tr>
            <td> B </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort2"/>
            </td>
          </tr>
          <tr>
            <td> C </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort3"/>
            </td>
          </tr>
          <tr>
            <td> D </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort4"/>
            </td>
          </tr>
          <tr>
            <td> E </td>
            <td>
              <bean:write name="Testverlauf"
                          property="aufgabe.wahlantwort5"/>
            </td>
          </tr>
        </table><p></p>
      </logic:notEqual>

      <table width="70%">
        <tr>
          <td align="center" width="20%">
            <label>
              A <input type="radio" name="antwort" value="A">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              B <input type="radio" name="antwort" value="B">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              C <input type="radio" name="antwort" value="C">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              D <input type="radio" name="antwort" value="D">
            </label>
          </td>
          <td align="center" width="20%">
            <label>
              E <input type="radio" name="antwort" value="E">
            </label>
```

```
            </td>
          </tr>
        </table>

      </logic:equal>

      <html:submit property="submit" value="Antwort abschicken"/>
    </html:form>

  </hr>

  Zeit seit Beginn: <bean:write name="Testverlauf" property="dauer"/>
  Sekunde(n).
  Es wurden bisher <bean:write name="Testverlauf"
                               property="gestellteAufgaben"/> Fragen gestellt,
  davon richtig: <bean:write name="Testverlauf" property="richtigeAufgaben"/>.

  <hr />

  <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

 </body>
</html>
```

Beispiel 8-27
aufgabenstellung.jsp

```
package at.ac.akhwien.mvc.formbeans;

import org.apache.struts.action.ActionForm;

public final class AntwortFormBean extends ActionForm
{
  // Eigenschaften
  private String antwort = "";

  // Methoden

  public void setAntwort (String antwort_neu)
  {
    this.antwort = antwort_neu;
  }

  public String getAntwort ()
  {
    return this.antwort;
  }
}
```

Beispiel 8-28
AntwortFormBean.jsp

```
package at.ac.akhwien.mvc.actions;

import java.io.*;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import org.apache.commons.beanutils.PropertyUtils;

import at.ac.akhwien.mvc.model.Aufgabenkatalog;
import at.ac.akhwien.mvc.model.Testverlauf;



/**
 * Korrektur
 *
 * @author Harald Kornfeil
 */

public final class Korrektur extends Action
{
  /**
  * Dient zur Abarbeitung eines HTTP Request.
  * Es wird die entsprechende Response generiert bzw. auf eine andere
  * Web-Komponente weitergeleitet, die sie dann erstellt.
  *
  * Es wird eine <code>ActionForward</code> Instanz zurückgegeben, die angiebt
  * wohin/wie weitergeleitet werden soll (oder <code>null</code> wenn die
  * Antwort bereits generiert wurde.
  *
  * @param mapping die ActionMapping die verwendet wird
  * @param form eine optionale ActionForm-Bean für diesen Request
  * @param request der HTTP request um den es geht
  * @param response die HTTP response die wir erzeugen
  *
  * @exception eine Exception falls Business-Logik eine Exception wirft
  */

  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) throws Exception
  {
    Testverlauf         testverlauf;
    HttpSession         session;
    String              antwort;
    String              weiter;

    session=request.getSession(false);

    testverlauf= (at.ac.akhwien.mvc.model.Testverlauf)
                                          session.getAttribute("Testverlauf");
    if (testverlauf==null)
    {
      return(mapping.findForward("index"));
    }

    antwort = (String) PropertyUtils.getSimpleProperty(form, "antwort");

    testverlauf.setAntwort(antwort);

    weiter=testverlauf.aufgabekorrigieren();

    return (mapping.findForward(weiter));

  }

}
```

Beispiel 8-29
korrektur.java

```
<%@page  session="true" %>


<%@ taglib prefix="html"  uri="/tags/struts-html" %>
<%@ taglib prefix="bean"  uri="/tags/struts-bean" %>

<html>
  <head>
    <title>Richtig</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>

  </head>
  <body bgcolor=white>

    Fragennummer: <bean:write name="Testverlauf" property="aufgabe.nr"/>
    <hr />
    <h1>Richtig!</h1>

    <p>Gegebene Antwort: <bean:write name="Testverlauf" property="antwort"/>,
       richtige Antwort: <bean:write name="Testverlauf"
                                     property="aufgabe.loesung"/> </p>

    <html:form method="GET" action="/aufgabe.do">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true" />
        diese Aufgabe nocheinmal stellen
      </p>
      <html:submit property="submit" value="weiter" />
    </html:form>

    <hr />
```

```
    Zeit seit Beginn: <bean:write name="Testverlauf" property="dauer"/>
    Sekunde(n).
    Es wurden bisher <bean:write name="Testverlauf"
                                 property="gestellteAufgaben"/> Fragen gestellt,
    davon richtig: <bean:write name="Testverlauf" property="richtigeAufgaben"/>.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>
    </hr>

  </body>
</html>
```

Beispiel 8-30
richtig.jsp

```
<%@page  session="true" %>


<%@ taglib prefix="html"  uri="/tags/struts-html" %>
<%@ taglib prefix="bean"  uri="/tags/struts-bean" %>

<html>
  <head>
    <title>Falsch</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>

  </head>
  <body bgcolor=white>

    Fragennummer: <bean:write name="Testverlauf" property="aufgabe.nr"/>
    <hr />
    <h1>Falsch</h1>

    <p>Gegebene Antwort: <bean:write name="Testverlauf" property="antwort"/>,
       richtige Antwort: <bean:write name="Testverlauf"
                                     property="aufgabe.loesung"/> </p>

    <html:form method="GET" action="/aufgabe.do">
      <p>
        <input type="checkbox" name="aufgabeWiederholen" value="true"
               checked="checked" />
        diese Aufgabe nocheinmal stellen
      </p>
      <html:submit property="submit" value="weiter" />
    </html:form>

    <hr />

    Zeit seit Beginn: <bean:write name="Testverlauf" property="dauer"/>
    Sekunde(n).
    Es wurden bisher <bean:write name="Testverlauf"
                                 property="gestellteAufgaben"/> Fragen gestellt,
    davon richtig: <bean:write name="Testverlauf" property="richtigeAufgaben"/>.

    <hr />
    <div class="rechts"> <a href="ende.jsp">Test vorzeitig beenden</a> </div>

  </body>
</html>
```

Beispiel 8-31
falsch.jsp

```
<%@page  session="true" %>

<%@taglib prefix="bean"  uri="/tags/struts-bean" %>

<jsp:useBean id="Testverlauf"
             class="at.ac.akhwien.mvc.model.Testverlauf" scope="session" >
  <% session.invalidate(); %>
  <jsp:forward page="index.jsp" />
</jsp:useBean>


<html>
  <head>
    <title></title>
  </head>
  <body bgcolor=white>
    <h1>Ende</h1>
    <h2>Auswertung:</h2>

    <p>
     Vom Computer wurden <bean:write name="Testverlauf" property="testumfang"/>
     verschieden Fragen aus einem Pool von
     <bean:write name="Testverlauf" property="katalog.anzahl"/>
     Aufgaben per Zufall ausgesucht.
    </p>

    <p>
     Die Dauer zur Beantwortung aller Fragen betrug
     <bean:write name="Testverlauf" property="dauer"/> Sekunde(n). Dies
     entspricht <%= Testverlauf.getDauer()/Testverlauf.getGestellteAufgaben() %>
     Sekunden pro Frage.
    </p>

    <p>
     Insgesammt wurden
     <bean:write name="Testverlauf" property="gestellteAufgaben"/> Fragen
     gestellt, von denen Sie
     <bean:write name="Testverlauf" property="richtigeAufgaben"/>  richtig
     beantwortet haben
     ( <%= 100*Testverlauf.getRichtigeAufgaben() /
                                        Testverlauf.getGestellteAufgaben() %> % )
    </p>


    <hr />
    <a href="index.jsp">zum Start</a>

  </body>
</html>

<% session.invalidate(); %>
```

Beispiel 8-32
ende.jsp

JSF-Version der Knoko-Lernprogramm-Webapplikation

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd" version="2.4">


  <display-name>Knoko Pruefungsvorbereitung</display-name>
  <description>
    Diese Webapplication dient zur Pruefungsvorbereitung fuer das
    Knochenkolloquium.
  </description>

  <context-param>
    <param-name>testumfang</param-name>
    <param-value>6</param-value>
  </context-param>

  <listener>
    <listener-class>at.ac.akhwien.mvc.model.SCInit</listener-class>
  </listener>


  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Beispiel 8-33
web.xml

```
<?xml version="1.0"?>

<!DOCTYPE faces-config
    PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">


<faces-config>

  <navigation-rule>
    <from-view-id>/aufgabenstellung.jsp</from-view-id>
    <navigation-case>
      <from-outcome>richtig</from-outcome>
      <to-view-id>/richtig.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>falsch</from-outcome>
      <to-view-id>/falsch.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/richtig.jsp</from-view-id>
    <navigation-case>
      <from-outcome>neu</from-outcome>
      <to-view-id>/aufgabenstellung.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/falsch.jsp</from-view-id>
    <navigation-case>
      <from-outcome>neu</from-outcome>
      <to-view-id>/aufgabenstellung.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>ende</from-outcome>
      <to-view-id>/ende.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>



  <managed-bean>
    <managed-bean-name>Aufgabenkatalog</managed-bean-name>
    <managed-bean-class>
      at.ac.akhwien.mvc.model.Aufgabenkatalog
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
      <property-name>dateiname</property-name>
      <value>#{applicationScope.dateiname}</value>
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Testverlauf</managed-bean-name>
    <managed-bean-class>
      at.ac.akhwien.mvc.model.Testverlauf
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>testumfang</property-name>
      <value>#{initParam.testumfang}</value>
    </managed-property>
    <managed-property>
      <property-name>katalog</property-name>
      <value>#{Aufgabenkatalog.referenz}</value>
    </managed-property>
  </managed-bean>


</faces-config>
```

Beispiel 8-34
faces-config.xml

```
<html>
  <head>
    <meta http-equiv="Refresh"
          content="0;URL=faces/index.jsp" />
  </head>
  <body bgcolor=white>
    Weiterleitung ...
  </body>
</html>
```

Beispiel 8-35
Ausschnitt aus index.html

```
<html>
  <head>
    <title>Home</title>
  </head>
  <body bgcolor=white>
    <h1>Willkommen</h1>
    <p>
    </p>
    <a href="aufgabenstellung.jsp">Start</a>.

    </hr>

  </body>
</html>
```

Beispiel 8-36
index.jsp

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>


<html>
  <head>
    <title>Knoko-Lernprogramm Frage</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    <f:view>

      <h:form>
        <h:outputText value="Fragennummer #{Testverlauf.aufgabe.nr}, " />
        <c:choose>
          <c:when test="${Testverlauf.aufgabe.modus==1}">
            <h:outputText value="Einfachauswahl (A)" />
            <h:panelGrid columns="1" styleClass="strich">

              <h:panelGrid columns="2"
                           columnClasses="links,rechts" >
                <h:outputText value="#{Testverlauf.aufgabe.fragenintro}"
                              escape="false" />
                <c:choose>
                  <c:when test="${Testverlauf.aufgabe.bildname!=''}">
                    <h:graphicImage value="#{Testverlauf.aufgabe.bildname}" />
                  </c:when >
                </c:choose>
              </h:panelGrid >

              <h:selectOneRadio id="wahlantworten"
                                value="#{Testverlauf.antwort}"
                                layout="pageDirection">
                <f:selectItem itemValue="A"
                           itemLabel="A .. #{Testverlauf.aufgabe.wahlantwort1}" />
                <f:selectItem itemValue="B"
                           itemLabel="B .. #{Testverlauf.aufgabe.wahlantwort2}" />
                <f:selectItem itemValue="C"
                           itemLabel="C .. #{Testverlauf.aufgabe.wahlantwort3}" />
                <f:selectItem itemValue="D"
                           itemLabel="D .. #{Testverlauf.aufgabe.wahlantwort4}" />
                <f:selectItem itemValue="E"
                           itemLabel="E .. #{Testverlauf.aufgabe.wahlantwort5}" />
              </h:selectOneRadio>
            </h:panelGrid >
          </c:when >

          <c:when test="${Testverlauf.aufgabe.modus==2}">
            <h:outputText value="Antwortkombinationsaufgabe (Kprim)" />
            <h:panelGrid columns="1" styleClass="strich">

              <h:panelGrid columns="1">
                <h:outputText value="#{Testverlauf.aufgabe.fragenintro}"
                              escape="false" />
                <h:outputText value="1 .. #{Testverlauf.aufgabe.wahlantwort1}"/>
                <h:outputText value="2 .. #{Testverlauf.aufgabe.wahlantwort2}"/>
                <h:outputText value="3 .. #{Testverlauf.aufgabe.wahlantwort3}"/>
              </h:panelGrid >

              <h:selectOneRadio id="wahlantworten"
                                value="#{Testverlauf.antwort}"
                                layout="lineDirection"
                                style="width:40em;table-layout:fixed;">
                <f:selectItem itemValue="A" itemLabel="A .. [ 1+2+3 ]" />
                <f:selectItem itemValue="B" itemLabel="B .. [ 1+2 ]" />
                <f:selectItem itemValue="C" itemLabel="C .. [ 2+3 ]" />
                <f:selectItem itemValue="D" itemLabel="D .. [ 1 ]" />
                <f:selectItem itemValue="E" itemLabel="E .. [ 3 ]" />
              </h:selectOneRadio>
            </h:panelGrid>
          </c:when >

          <c:when test="${Testverlauf.aufgabe.modus==3}">
            <h:outputText value="Zuordnungsfrage (B)" />
            <h:panelGrid columns="1" styleClass="strich">
              <h:panelGrid columns="2"
                           columnClasses="links,rechts"
                           rowClasses="zeile">
                <h:outputText value="#{Testverlauf.aufgabe.fragenintro}"
                              escape="false" />
                <c:choose>
                  <c:when test="${Testverlauf.aufgabe.bildname!=''}">
                    <h:graphicImage value="#{Testverlauf.aufgabe.bildname}" />
                    <f:facet name="footer">
                      <h:outputText value="Wenn der gefragte Begriff durch
                                           'A' - 'D' nicht bezeichnet ist,
                                           ist 'E' zu wählen." />
                    </f:facet>
                  </c:when >
                </c:choose>
              </h:panelGrid >

              <c:choose>
                <c:when test="${Testverlauf.aufgabe.bildname!=''}">
                  <h:selectOneRadio id="wahlantworten"
                                    value="#{Testverlauf.antwort}"
                                    layout="lineDirection" style="width:20em">
                    <f:selectItem itemValue="A" itemLabel="A " />
                    <f:selectItem itemValue="B" itemLabel="B " />
                    <f:selectItem itemValue="C" itemLabel="C " />
                    <f:selectItem itemValue="D" itemLabel="D " />
                    <f:selectItem itemValue="E" itemLabel="E " />
                  </h:selectOneRadio>
                </c:when >
                <c:otherwise>
                  <h:selectOneRadio id="wahlantworten"
                                    value="#{Testverlauf.antwort}"
                                    layout="pageDirection">
                    <f:selectItem itemValue="A"
                         itemLabel="A .. #{Testverlauf.aufgabe.wahlantwort1}" />
                    <f:selectItem itemValue="B"
                         itemLabel="B .. #{Testverlauf.aufgabe.wahlantwort2}" />
                    <f:selectItem itemValue="C"
                         itemLabel="C .. #{Testverlauf.aufgabe.wahlantwort3}" />
                    <f:selectItem itemValue="D"
                         itemLabel="D .. #{Testverlauf.aufgabe.wahlantwort4}" />
                    <f:selectItem itemValue="E"
                         itemLabel="E .. #{Testverlauf.aufgabe.wahlantwort5}" />
                  </h:selectOneRadio>
                </c:otherwise>
              </c:choose>
            </h:panelGrid>
          </c:when >
        </c:choose>

        <h:commandButton action="#{Testverlauf.aufgabekorrigieren}"
                         value="Abschicken" />
      </h:form>

      <h:panelGrid columns="1" styleClass="strich">
        <h:panelGroup>
          <h:outputText value="Zeit seit Beginn #{Testverlauf.dauer} sec. " />
          <h:outputText value="Es wurden bisher #{Testverlauf.gestellteAufgaben}
                               Fragen gestellt, " />
          <h:outputText value="davon richtig: #{Testverlauf.richtigeAufgaben}"/>
        </h:panelGroup>
      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="strich" columnClasses="rechts">
          <h:commandLink action="ende">
            <h:outputText value="Test vorzeitig beenden." />
          </h:commandLink>
        </h:panelGrid >
      </h:form>

    </f:view>

  </body>
</html>
```

Beispiel 8-37
aufgabenstellung.jsp

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>


<html>
  <head>
    <title>Richtig</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
```

```
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    <f:view>

      <h:outputText value="Fragennummer #{Testverlauf.aufgabe.nr}" />

      <h:panelGrid columns="1" styleClass="alles" columnClasses="strich">
        <h:panelGroup>
          <h:outputText styleClass="h1" value="Richtig!" />

          <h:form>
            <h:panelGrid columns="1" columnClasses="doppelt" >
              <h:outputText value="Gegebene Antwort: #{Testverlauf.antwort},
                              richtige Antwort: #{Testverlauf.aufgabe.loesung}" />
              <h:panelGroup>
                <h:selectBooleanCheckbox id="wiederholen"
                                         value="#{Testverlauf.aufgabeWiederholen}" />
                <h:outputLabel for="wiederholen">
                  <h:outputText value="diese Aufgabe nocheinmal stellen" />
                </h:outputLabel>
              </h:panelGroup>

              <h:commandButton action="#{Testverlauf.neueAufgabe}"
                               value="Weiter" />
            </h:panelGrid >
          </h:form>
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt" >
          <h:outputText value="Zeit seit Beginn:
                               #{Testverlauf.dauer} Sekunde(n)." />
          <h:outputText value="Es wurden bisher #{Testverlauf.gestellteAufgaben}
                               Fragen gestellt, " />
          <h:outputText value="davon richtig: #{Testverlauf.richtigeAufgaben}"/>
        </h:panelGroup>

      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="alles" rowClasses="strich"
                                                    columnClasses="rechts">

          <h:commandLink action="ende">
            <h:outputText value="Test vorzeitig beenden." />
          </h:commandLink>
        </h:panelGrid >
      </h:form>

    </f:view>
  </body>
</html>
```

Beispiel 8-38
richtig.jsp

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>


<html>
  <head>
    <title>Falsch</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>
  <body>

    <f:view>

      <h:outputText value="Fragennummer #{Testverlauf.aufgabe.nr}" />

      <h:panelGrid columns="1" styleClass="alles" columnClasses="strich">
        <h:panelGroup>
          <h:outputText styleClass="h1" value="Falsch!" />
```

```
          <h:form>
            <h:panelGrid columns="1" columnClasses="doppelt" >
              <h:outputText value="Gegebene Antwort: #{Testverlauf.antwort},
                              richtige Antwort: #{Testverlauf.aufgabe.loesung}" />
              <h:panelGroup>
                <h:selectBooleanCheckbox id="wiederholen"
                                         value="#{Testverlauf.aufgabeWiederholen}" />
                <h:outputLabel for="wiederholen">
                  <h:outputText value="diese Aufgabe nocheinmal stellen" />
                </h:outputLabel>
              </h:panelGroup>

              <h:commandButton action="#{Testverlauf.neueAufgabe}"
                               value="Weiter" />
            </h:panelGrid >
          </h:form>
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt" >
          <h:outputText value="Zeit seit Beginn: #{Testverlauf.dauer}
                               Sekunde(n)." />
          <h:outputText value="Es wurden bisher #{Testverlauf.gestellteAufgaben}
                               Fragen gestellt, " />
          <h:outputText value="davon richtig: #{Testverlauf.richtigeAufgaben}"/>
        </h:panelGroup>

      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="alles" rowClasses="strich"
                                                    columnClasses="rechts">

          <h:commandLink action="ende">
            <h:outputText value="Test vorzeitig beenden." />
          </h:commandLink>
        </h:panelGrid >
      </h:form>

    </f:view>
  </body>
</html>
```

Beispiel 8-39
falsch.jsp

```
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core" %>

<%@taglib prefix="sess" uri="http://jakarta.apache.org/taglibs/session-1.0" %>

<html>
  <head>
    <title>Ende - Testzusammenfassung</title>
    <style type="text/css">
    <!--
      .links   { text-align:left;}
      .rechts  { text-align:right;}
      .alles   { width:100% }
      .zeile   { vertical-align:top;}
      .h1      { font-size: 200%; font-weight:bold; line-height:200% }
      .h2      { font-size: 150%; font-weight:bold; line-height:200% }
      .strich  { border-top:2px inset silver; width:100%;}
      .doppelt { line-height:200% }
    -->
    </style>
  </head>

  <body>

    <f:view>

      <h:panelGrid columns="1" >
        <h:outputText value="Ende" styleClass="h1" />
        <h:outputText value="Auswertung:" styleClass="h2" />

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Vom Computer wurden #{Testverlauf.testumfang}
                               verschieden Fragen " />
          <h:outputText value="aus einem Pool von #{Testverlauf.katalog.anzahl}
                               Aufgaben per Zufall ausgesucht." />
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Die Dauer zur Beantwortung aller Fragen betrug
                               #{Testverlauf.dauer} Sekunde(n). " />
          <h:outputText value="Dies entspricht "/>
          <h:outputText
                    value="#{Testverlauf.dauer/Testverlauf.gestellteAufgaben}">
            <f:convertNumber maxFractionDigits="2" />
          </h:outputText >
          <h:outputText value=" Sekunden pro Frage." />
```

```
        </h:panelGroup>

        <h:panelGroup styleClass="doppelt">
          <h:outputText value="Insgesamt wurden#{Testverlauf.gestellteAufgaben}
                                                          Fragen gestellt, " />
          <h:outputText value="von denen Sie #{Testverlauf.richtigeAufgaben}
                                                  richtig beantwortet haben ( " />
          <h:outputText value="#{ 100*Testverlauf.richtigeAufgaben /
                                              Testverlauf.gestellteAufgaben}">
            <f:convertNumber maxFractionDigits="2" />
          </h:outputText >
          <h:outputText value=" % )." />
        </h:panelGroup>

      </h:panelGrid >

      <h:form>
        <h:panelGrid columns="1" styleClass="strich" >
          <h:outputLink value="index.jsp" >
            <h:outputText value="zum Start" />
          </h:outputLink>
        </h:panelGrid >
      </h:form>

    </f:view>

  </body>
</html>

<sess:invalidate/>
```

Beispiel 8-40
ende.jsp

Anhang 3 Dateiformat der Textdatei knokofragen.txt

Dateiformat

Es handelt sich um eine Textdatei in der jede Zeile einem Datensatz (= eine Aufgabe) entspricht. Die einzelnen Datenfelder sind durch Tabulator-Zeichen getrennt.

Fragennummer
Fragenmodus/Fragentyp
Fragenintro
Richtige Antwort
Fragestellen
Wahlantwort1
Wahlantwort2
Wahlantwort3
Wahlantwort4
Wahlantwort5
Bilddateiname
Zusätzliche Informationen

Beispiel

```
1  1  Farbe?    D  1  Rot   Blau  Grün  Weiß  Schwarz  Schnee.jpg  Schnee
2  2             C  1  Nord  Süd   Ost   West  keines   Kompass.gif Antarktis
5  1  Franzose? A  1  Opel  Ford  Kia   VW    Renault              Aussprache
3  ...
```

Beispiel 8-41
Beispiel für eine Fragendatei