1 Einführung

1.1 Zweck des Dokuments

Dieses Dokument dient in erster Linie mir selbst, zur Vorbereitung auf die Zertifizierung zum Sun Certified Web Developer. Sollte es anderen auch nützen, so freut es mich.

Da bis jetzt (Mitte Juli 04) noch kein Buch zur Vorbereitung von SCWCD 1.4 existiert, habe ich anhand der JSP 2.0 und Servelt 2.4 und den unten aufgelisteten Büchern, das Wichtigste nach den Objectives geordnet und herausgeschrieben. Ich bin selbst überrascht, wie gross das Dokument im Laufe der Zeit geworden ist.

Ich bin gerade in der Endphase der Vorbereitung, und entdecke bei der Wiederholung noch sehr viele Fehler in diesem Dokument. Ich veröffentliche trotzdem dieses Dokument im Internet. Je mehr Interessierte dieses Lesen, desto eher werden die Fehler ausgemerzt. Falls einem der Leser Fehler auffällt, bitte ich diese mir zu mailen .

Ein anderes Ziel ist es die von mir geschriebene Software CoCoDiL zu testen. Es ist schon möglich ohne Programmierkenntnisse Html Dokumente und PDF-Dateien zu erzeugen. Das Ziel ist es aber, dies als eine Art Wiki-Server zu implementieren. Die Kurse sollten von Jedermann (natürlich auch Frau) vom Internet heraus editierbar und erweiterbar sein. Der nächste Schritt sind Links zu einfachen Übungen und Diskussionen einzubauen. Die Übungen sollten in einer einfachen Sprache auch über das Internet editierbar sein.

Den meisten die die Zertifizierung machen geht es wahrscheinlich wie mir. Ich möchte in diesem Bereich arbeiten, kann aber keine praktische Erfahrung nachweisen. Die Zertifizierung erhöht sicherlich die Chancen auf den Arbeitsmarkt, kann aber praktische Erfahrung nicht ersetzen.

Aber selbst für Leute die praktische Erfahrung haben, kann die Zertifizierung sinnvoll sein. Mir ist z.B aufgefallen das viele Programmierer die schon mehrjährige Java Erfahrung haben, nicht den Unterschied zwischen protected und der "default" Sicht kennen. Der Grund liegt daran, das in der IT-Branche viele Firmen nicht die finanziellen Möglichkeiten haben ihre Leute auszubilden, und innerhalb der Firma kein Wissensaustausch stattfindet. Als ich letztes Jahr einige Leute gefunden hatte, um die Zertifizierung zum Programmer zu machen; haben auch Leute daran teilgenommen die mehrere Jahre praktische Erfahrung hatten. Sie wollten die Zertifizierung selbst gar nicht machen, haben aber durch die Nacharbeitung der Theorie nach eigenen Aussagen sehr profitiert, (und ich von deren praktischen Erfahrung).

1.2 Vorbereitungstips zur Zertifizierung

Die beste Vorbereitung sind Mock Examen. Ich habe mir von Whizlabs den SCWCD 1.4 Exam Simulator gekauft, da im Internet noch nicht sehr viele vorhanden sind. Dieser SCWCD 1.4 Exam Simulator macht einen sehr guten Eindruck und hilft mir enorm.Bei nicht Bestehen der Zertifizierung bekommt man sein Geld zurück. (Ich bekomme kein Geld von Whizlabs für diese Reklame). Ich glaube ohne Mock Examen ist die Zertifizierung fast nicht möglich.

Die Mock Examen sind nur lösbar, wenn man vorher sich den theoretischen Hintergrund bearbeitet hat. Eine Literaturliste ist weiter unten.

Das Examen zum SCWCD 1.4 gibt es noch nicht sehr lange, deshalb findet man noch relativ

wenig Hilfen im Internet. Whizlabs selbst hat eine Auflistung von Foren, und Diskussionen über SCWCD.

Unbedingt zu empfehlen ist, diese Kenntnisse praktisch umzusetzen. Ich bin gerade dabei CoCoDiL als WebServer zu implementieren.

Sehr Vorteilhaft ist sich mit anderen Leuten zusammenzuschliessen. Für die Zertifizierung muss man einige Stunden pro Woche abzweigen. Die Diziplin regelmaessig auch nach der Arbeit sich nocheinmal hinzusetzen; geht am leichtesten mit Mitlernern. In der Schlussphase der Vorbereitung man wohl mehr als einige Stunden pro Woche aufwenden. Hier wird man wohl 1-2 Wochen Urlaub opfern müssen.

Ich habe neben der Zertifizierungsgebühr insgesamt noch einmal ca. 200 Euro für MockExamen und Bücher ausgegeben. Dies liegt aber auch daran das die SCWCD in Version 1.4 noch relativ neu ist. Wer die ältere Version 1.3 machen will, kommt wohl alleine mit dem Buch *The Scwcd Exam Study Kit* zurecht.

1.3 Verwendete Literatur

Ich habe für dieses Dokument folgende Literatur verwendet:

- The Scwcd Exam Study Kit: Java Web Component Development Certification Link

Dies ist wirklich ein hervorragendes Buch. Es bereitet aber auf SCWCD 1.3 und nicht auf SCWCD 1.4 vor. Für die Zertifizierung für Version 1.3 braucht man keine weitere Literatur, aber auch für die aktuelle Version ist es sehr hilfreich (wenn auch in 1.4 viele neue Konzepte hinzugekommen sind). Es wird erwartet das eine neue Auflage für die Version 1.4 erscheinen wird. Es ist auch eine CD mit einigen Mock Examen dabei.

- Java Server Pages 3.Auflage Link

Dies ist ein Buch das sehr schnell in die Praxis führt. Für dijenigen die JSP schnell erlernen und anwenden wollen ist das Buch sehr zu empfehlen. Für die Zertifizierung selbst ist der Anhang mit dem Referenzteil sehr hilfreich. Ansonsten hat man das Problem das der Aufbau des Buches eine total andere Struktur hat, wie durch die Objectives vorgegeben. Dies Buch basiert auf den neusten JSP 2.0 Standard.

- Einstieg in JavaServerPages 2.0 Link

Ich muss gestehen dass ich am Anfang sehr entäuscht war, was teilweise am schlechten Druck lag. Aber schliesslich habe ich immer öfters zu diesem Buch gegriffen. Der grösste Vorteil ist, daß es auf Deutsch ist. Die Beispiele sind übersichtlich gehalten. Auf CD befindet sich TomCat 5.0, die Beispiele die JSP 2.0, Servlets 2.4 und JSTL Spezifikation. Auch dieses Buch kann ich empfehlen.

- Design-Patterns von Sun

Hier werden die wichtigsten J2EE Design Patterns erläutert. Diese Information fehlt weitgehend in den angegebenen Büchern. (The Scwcd Exam Study Kit erläutert alle die für die 1.3 Version benötigt werden).

- Dies soll andere existierende Bücher nicht abwerten, ich kann nur das empfehlen was ich

selber gelesen habe.

2 Exam Objectives

1. The Servlet Technology Model


For each of the HTTP Methods (such as GET, POST, HEAD, and so on) describe the purpose of the method and the technical characteristics of the HTTP Method protocol, list triggers that might cause a Client (usually a Web browser) to use the method; and identify the HttpServlet method that corresponds to the HTTP Method.

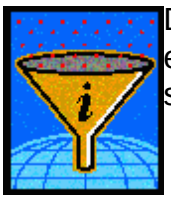

Using the HttpServletRequest interface, write code to retrieve HTML form parameters from the request, retrieve HTTP request header information, or retrieve cookies from the request.


Using the HttpServletResponse interface, write code to set an HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL, or add cookies to the response.


Describe the purpose and event sequence of the servlet life cycle: (1) servlet class loading, (2) servlet instantiation, (3) call the init method, (4) call the service method, and (5) call destroy method.

2. The Structure and Deployment of Web Applications


Construct the file and directory structure of a Web Application that may contain (a) static content, (b) JSP pages, (c) servlet classes, (d) the deployment descriptor, (e) tag libraries, (f) JAR files, and (g) Java class files; and describe how to protect resource files from HTTP access.


Describe the purpose and semantics for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

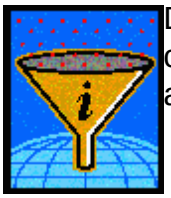

Construct the correct structure for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.


Explain the purpose of a WAR file and describe the contents of a WAR file, how one may be constructed.

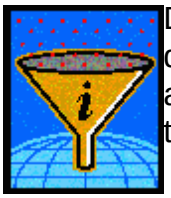
3. The Web Container Model


For the ServletContext initialization parameters: write servlet code to access initialization parameters; and create the deployment descriptor elements for declaring initialization parameters.


For the fundamental servlet attribute scopes (request, session, and context): write servlet code to add, retrieve, and remove attributes; given a usage scenario, identify the proper scope for an attribute; and identify multi-threading issues associated with each scope.


Describe the Web container request processing model; write and configure a filter; create a request or response wrapper; and given a design problem, describe how to apply a filter or a wrapper.


Describe the Web container life cycle event model for requests, sessions, and web applications; create and configure listener classes for each scope life cycle; create and configure scope attribute listener classes; and given a scenario, identify the proper attribute listener to use.


Describe the RequestDispatcher mechanism; write servlet code to create a request dispatcher; write servlet code to forward or include the target resource; and identify and describe the additional request-scoped attributes provided by the container to the target resource.

4. Session Management


Write servlet code to store objects into a session object and retrieve objects from a session object.


Given a scenario describe the APIs used to access the session object, explain when the session object was created, and describe the mechanisms used to destroy the session object, and when it was destroyed.


Using session listeners, write code to respond to an event when an object is added to a session, and write code to respond to an event when a session object migrates from one VM to another.


Given a scenario, describe which session management mechanism the Web container could employ, how cookies might be used to manage sessions, how URL rewriting might be used to manage sessions, and write servlet code to perform URL rewriting.

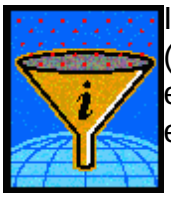
5. Web Application Security


Based on the servlet specification, compare and contrast the following security mechanisms: (a) authentication, (b) authorization, (c) data integrity, and (d) confidentiality.


In the deployment descriptor, declare a security constraint, a Web resource, the transport guarantee, the login configuration, and a security role.


Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and given a scenario, select an appropriate type.

6. The JavaServer Pages (JSP) Technology Model


Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.


Write JSP code that uses the directives: (a) 'page' (with attributes 'import', 'session', 'contentType', and 'isELIgnored'), (b) 'include', and (c) 'taglib'.


Write a JSP Document (XML-based document) that uses the correct syntax.


Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the jspInit method, (6) call the _jspService method, and (7) call the jspDestroy method.


Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.


Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language.


Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the include directive or the jsp:include standard action).

7. Building JSP Pages Using the Expression Language (EL)


Given a scenario, write EL code that accesses the following implicit variables including: pageScope, requestScope, sessionScope, and applicationScope, param and paramValues, header and headerValues, cookie, initParam and pageContext.


Given a scenario, write EL code that uses the following operators: property access (the '.' operator), collection access (the '[]' operator).


Given a scenario, write EL code that uses the following operators: aritmetic operators, relational operators, and logical operators.


Given a scenario, write EL code that uses a function; write code for an EL function; and configure the EL function in a tag library descriptor.

8. Building JSP Pages Using Standard Actions


Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: 'id', 'scope', 'type', and 'class'), jsp:getProperty, and jsp:setProperty (with all attribute combinations).


Given a design goal, create a code snippet using the following standard actions: jsp:include, jsp:forward, and jsp:param.

9. Building JSP Pages Using Tag Libraries


For a custom tag library or a library of Tag Files, create the 'taglib' directive for a JSP page.


Given a design goal, create the custom tag structure in a JSP page to support that goal.


Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.

10. Building a Custom Tag Library


Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.


Using the PageContext API, write tag handler code to access the JSP implicit variables and access web application attributes.


Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.


Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.


Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.

11. J2EE Patterns


Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.


Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.

3 The Servlet Technology Model

3.1 Http Methods


For each of the HTTP Methods (such as *GET*, *POST*, *HEAD*, and so on) describe the purpose of the method and the technical characteristics of the HTTP Method protocol, list triggers that might cause a Client (usually a Web browser) to use the method; and identify the *HttpServlet* method that corresponds to the HTTP Method.

3.1.1 Überblick über die Http Methoden

Der Client (i.a. ein Webbrowser) sendet Anfragen an das Servlet und bekommt von diesem Servlet eine Antwort. Ein Servlet ist ein Programm, das auf einem i.a. entfernten Server liegt. Das Servlet liest die Anfragen von seinen Clienten, verarbeitet sie und sendet das Ergebnis i.a an den Clienten zurück.

Damit Client und Servlet miteinander kommunizieren können, müssen sie eine gemeinsame Sprache verstehen. Diese gemeinsame Sprache ist durch das Hypertext Transfer Protocol definiert.

Eine HttpNachricht besteht aus einer

1. Startzeile
2. Header
3. CRLF (Carriage Return und LineFeed also einen Zeilenumbruch)
4. Nachrichtenrumpf

Das erste Wort in der ersten Zeile besteht aus einer Http-Methode. Folgende Http-Methoden sind definiert:

1. GET - Liest Informationen von Servlet
2. POST - Gibt dem Servlet Daten zur Verarbeitung
3. PUT - Gibt dem Servlet Daten zum Speichern
4. HEAD - Liest i.a. nur MetaInformation vom servlet
5. DELETE - Dient zum Löschen von Daten auf dem Server
6. OPTiONS - Fragt dem Servlet nach welche Methoden es versteht.
7. TRACE - Dient i.a. zum Debuggen

Ein Servlet wird in Java durch eine Unterklasse der abstrakten Klasse *HttpServlet* implementiert. Je nach Http Methode wird in der Klasse *HttpServlet* eine spezielle Methode ausgeführt.

```
protected void doXXX(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException

//  XXX steht ist Platzhalter Get,Post, Put, Head, Options, Trace
```

req enthält die Anfrage die vom Clienten kommt, die Antwort wird in *resp* geschrieben. Ein einfaches Beispiel:

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

public class DisplayServlet extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws IOException, ServletException {
                resp.setContentType("text/html");    // Ausgabe ist als Html File zu interpretieren
                PrintWriter out = resp.getWriter();
                out.println("<html><head><title>Display Information");
                out.println("</title></head><body>");
                out.println("Hello, World");
                out.println("</body></html>");
        }
}
```

Es kann nicht garantiert werden dass eine *doGet()* nur Informationen liest und keine Verarbeitung macht, es liegt in der Verantwortung des Programmiers diese Konventionen einzuhalten.

3.1.2 die doGet Methode

- Die Get Methode sollte keine Seiteneffekte haben, d.h keine Änderungen am Server
- Die Get Methode sollte wiederholbar sein, und jeweils dasselbe Ergebnis zurückliefern
- Die Daten die Get Methode sendet, sind im URL Feld des Browsers sichtbar.
- Mit der get Methode kann man nur Ascii Textdaten versenden.
- Die meisten Browser limitieren die Anzahl der Zeichen die mitgegeben werden dürfen.
- Paramter werden mit einem '&' getrennt.

```
name1=value1&name2=value2&name3=value3
```

Wie wird die Methode getriggert?

- Eingabe einer URL im AdressFeld des Browser.
- Mausclick auf einen Verweis.

- Klick auf ein HTML-Formular das die get-Methode verwendet.

Beispiel:

Beachte beim Ausführen des Formulars, dass die Parameter firstName und lastName in der URL Adresse als Parametername enthalten sind.

```
<form action="http://www.klausmeucht.de" method="GET">
        Vorname: <input type="text" name="firstName"><p>
        Name:    <input type="text" name="lastName"><p>
        <input type="submit" value="Display">
</form>
```

<form action="http://www.klausmeucht.de" method="GET"> Vorname: <input type="text" name="firstName"><p> Name: <input type="text" name="lastName"><p> <input type="submit" value="Ausführen"> </form>

3.1.3 die doPost Methode

- Die Post Methode dient dazu Daten an einen Server zu senden und diese zu bearbeiten
- Typische Anwendung ist in eine newsgroup zu posten, Daten in eine Datenbank zu schreiben.
- Es wird nicht erwartet, dass die Methode sicher ist, d.h. keine Änderungen am Servlet zulässt
- Es können ASCII und Binary Daten gesendet werden
- Die Grösse der gesendeten Daten ist nicht begrenzt
- Die gesendeten Daten sind nicht im Browser sichtbar

Wie wird die Methode getriggert?

- Klick auf ein HTML-Formular das die post-Methode verwendet.

Beispiel:

Beachte beim Ausführen des Formulars, dass die Parameter firstName und lastName in der URL nicht zu sehen ist.

```
<form action="http://www.klausmeucht.de" method="POST">
        Vorname: <input type="text" name="firstName"><p>
        Name:    <input type="text" name="lastName"><p>
        <input type="submit" value="Display">
</form>
```

<form action="http://www.klausmeucht.de" method="POST"> Vorname: <input type="text" name="firstName"><p> Name: <input type="text" name="lastName"><p> <input type="submit" value="Ausführen"> </form>

3.1.4 die doPut Methode

- Die Put Methode dient dazu Daten auf einen Server zu senden.
- Eine typische Anwendung wäre die Daten mit ftp an einen Server zu senden.
- Im Gegensatz zur POST Methode steht nicht die Verarbeitung der Daten im Vordergrund, sondern es die Daten werden lediglich gespeichert
- Es können ASCII und Binary Daten gesendet werden
- Die Grösse der gesendeten Daten ist nicht begrenzt
- Die gesendeten Daten sind nicht im Browser sichtbar

Wie wird diese Methode getriggert?

- Klick auf ein HTML-Formular das die put-Methode verwendet.

3.1.5 die doHead Methode

- Die Head Methode ist eine eingeschränkte GET-Methode. Als Antwort wird lediglich der Header zurückgegeben.
- Dient z.B zum Synchronisieren von Dateien. Es wird z.B im Header erstmals Attribut lastModified ausgelesen, erst falls Information von Client und Server nicht übereinstimmt, kann die gesamte Information ausgelesen werden.
- Es gelten dieselben Einschränkungen wie bei der doGet Methode.

3.1.6 die doDelete Methode

- Diese Methode wird sehr selten verwendet.
- Der Server wird angehalten, eine Information zu löschen.

3.1.7 die doOptions Methode

- Diese Methode wird noch seltener verwendet.
- Der Server wird gefragt welche HttpMethoden er unterstützt.

3.1.8 die doTrace Methode

- Dient i.a. aussschliesslich zum Debuggen.
- Die doTrace Methode wird ein Programmierer i.a. nicht überschreiben.

3.2 HttpServletRequest Interface

Using the HttpServletRequest interface, write code to retrieve HTML form parameters from the request, retrieve HTTP request header information, or retrieve cookies from the request.

Hier die Methoden des Interfaces die für dieses Objective interessant ist.

```
public interface ServletRequest {
    public String getParameter(String name);
    public Enumeration getParameterNames();
    public String[] getParameterValues(String name);
    public Map getParameterMap();
    ...
}
```

```
public interface HttpServletRequest extends ServletRequest {
    public Cookie[] getCookies();
    public long getDateHeader(String name);
    public String getHeader(String name);
    public Enumeration getHeaders(String name);
    public Enumeration getHeaderNames();
    public int getIntHeader(String name);
    ...
}
```

3.2.1 Auslesen der Parameter

Parameter werden als key value Paare gespeichert. Unter einem Schlüssel können auch mehrere Werte gespeichert sein.

- public String getParameter(String name);

Es wird der Wert des Parameters *name* mitgegeben, oder *null*. Falls unter dem Paramter *name* mehrere Werte gespeichert sind, wird der erste zurückgegeben.

- public Enumeration getParameterNames();

Liefert die Schlüssel aller Parameter der Anfrage zurück. Hat die Anfrage kein Parameter, so wird eine leere *Enumeration* zurückgegeben.

- public String[] getParameterValues(String name);

Diese Methode sollte verwendet werden, falls mehrere Werte unter dem Parameter *name* gespeichert sind. Gibt falls es den Parameter *name* nicht gibt *null* zurück.

- public Map getParameterMap();

Gibt eine *java.util.map* aller Parameter mit. Diese *map* kann nicht verändert werden. Die Schlüssel sind als Strings die Parameter werden jeweils als Array von Strings mitgegeben.

3.2.2 Auslesen des Headers

- public String getHeader(String name);

Es wird der Wert des Headers *name* mitgegeben, oder *null*. Falls unter dem Header *name* mehrere Werte gespeichert sind, wird der erste zurückgegeben.

- public Enumeration getHeaders(String name);

Diese Methode sollte verwendet werden, falls mehrere Werte unter dem Header *name* gespeichert sind. Gibt falls es den Header *name* nicht gibt *null* zurück.

- public Enumeration getHeaderNames();

Liefert die Schlüssel aller Header der Anfrage zurück. Hat die Anfrage kein Parameter, so wird eine leere *Enumeration* zurückgegeben.

- public int getIntHeader(String name);

Gibt es unter dem Parameter *name* keinen Header, wird -1 zurückgegeben. Ist der Wert nicht als Integer konvertierbar, so wird eine *NumberFormatException* geworfen.

- public long getDateHeader(String name);

Es wird die Anzahl der Millisekunden seit 1.1.1970 zurückgegeben. Gibt es unter dem Parameter *name* keinen Header, wird -1 zurückgegeben. Ist der Wert nicht als long konvertierbar, so wird eine *IllegalArgumentException* geworfen.

3.2.3 Auslesen der Cookies

Unter Cookies werden - meist kleine- Datenstückchen im Textformat verstanden, die ein Webserver bei der Antwort auf eine Anfrage auf der Festplatte eines Clienten ablegt.

Ein Cookie hat mindestens einen Namen und einen Wert, und muss beim Constructor mitgegeben werden.

```
public Cookie(String name, String value)
```

Ein Cookie kann unter anderem folgende Attribute enthalten:

- comment - beschreibt die Funktion eines Cookies
- max-age - Anzahl von Millisekunden in der das Cookie gültig ist. -1 beschränkt den Cookie auf einen Zeitpunkt bis der Browser geschlossen ist.
- path - gibt den URL Pfad der Seiten an, die das Cookie lesen dürfen. Beachte es dürfen 2 Cookies mitdenselben Namen aber unterschiedlichen Pfaden abgespeichert werden.
- domain - legt durch ein Zeichenmuster fest, dass der Server, zu dem das Cookie zurückgesendet wird, Mitglied der angegebenen Domain sein muss.
- secure - gibt an, dass das Cookie nur für SSL Verbindungen verwendet werden darf.
- version - Protokollversion mit der das Cookie übereinstimmet

Die Cookies die das Servlet lesen darf bekommt man durch:

- public Cookie[] getCookies();

Sind keine Cookies vorhanden wird der Wert *null* zurückgegeben.

3.3 Das HttpServletResponse Interface

Using the HttpServletResponse interface, write code to set an HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL, or add cookies to the response.

Hier die Methoden des Interface, die für dieses Objective interessant sind

```
public interface ServletResponse {
   public PrintWriter getWriter() throws IOException;
   public ServletOutputStream getOutputStream() throws IOException;
   public void setContentType(String type);
   ...
}
```

```
public interface HttpServletResponse extends ServletResponse{
   public void addCookie(Cookie cookie);
   public boolean containsHeader(String name);
   public void addHeader(String name, String value);
   public void setHeader(String name, String value);
   public void addDateHeader(String name, long date);
   public void setDateHeader(String name, long date);
   public void addIntHeader(String name, int value);
   public void setIntHeader(String name, int value);
   public void sendRedirect(String location) throws IOException;
   ...
}
```

3.3.1 Setzen der Headers

- public boolean containsHeader(String name);

Überprüft ob ein Header mit dem Parameter *name* gesetzt ist.

- public void setHeader(String name, String value);
- public void setDateHeader(String name, long date);
- public void setIntHeader(String name, int value);

Diese Methoden setzen einen Header. Existieren schon Header mit denselben Namen, werden diese überschrieben.

- public void addHeader(String name, String value);
- public void addDateHeader(String name, long date);
- public void addIntHeader(String name, int value);

Diese Methoden setzen einen Header, überschreiben aber nicht existierende. Sie werden benutzt um Header mit mehreren Werten zu erzeugen.

3.3.2 Textuelle und Binäre Ausgabe

- setContentType(String)

Mit *setContentType* wird mitgeteilt welche Art von Daten übertragen werden, und wie sie zu bearbeiten sind, entsprechend der MIME Typs

Beispiele:

```
setContentType("text/html");   // Default Einstellung
setContentType("text/plain");
setContentType("text/xml");
setContentType("application/jar");
```

setContentType(String) sollte am Anfang einer Ausgabe stehen.

- public PrintWriter getWriter() throws IOException;

Diese Methode wird für Ausgaben mit ASCII Texten benutzt.

Beispiel:

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

public class KinoServlet extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws IOException, ServletException {
                resp.setContentType("text/html");    // Ausgabe ist als Html File zu interpretieren
                PrintWriter out = resp.getWriter();
                out.println("<html>");
                out.println("<title>Der Schuh des Manitu</title>");
                ...
        }
}
```

Achtung: Man kann nicht dies nicht mit einem OutputStream vermischen

- public ServletOutputStream getOutputStream() throws IOException;

Diese Methode wird verwendet um Binärdaten auszugeben. Auch hier ein Beispiel:

```
public void doGet( HttpServletRequest req,
                   HttpServletResponse res)
           throws ServletException, IOException
```

```
{
        res.setContentType("application/jar");

        File f = new File("test.jar");
        byte[] bytearray = new byte[(int) f.length()];
        FileInputStream is = new FileInputStream(f);
        is.read(bytearray);

        OutputStream os = res.getOutputStream();
        os.write(bytearray);
        os.flush();
}
```

Der Aufruf von flush committed die Ausgabe, danach kann nichts mehr ausgegeben werden. *getOutputStream* und *getWriter* können nicht gemeinsam benutzt werden.

3.3.3 Umleitung einer Http Umfrage an eine andere URL

- public void sendRedirect(String location) throws IOException;

Mit dieser Methode kann eine Anfrage an eine andere URL umgeleitet werden. Location kann auch eine relative Adresse sein.

Beginnt diese Adresse mit einem Slash /, so wird location an die URL des Servlet Containers angehängt. Beginnt diese nicht mit einem Slash so wird die Adresse relativ zur URL des aktuellen Servlets angenommen.

Die Methode sendRedirect ist nicht transparent für den Browser. D.h der Browser wird aufgefordert die neue URL zu laden. Vorhergehende Ausgabenanweisungen werden ignoriert. Der Versuch nach einer Ausgabe die schon committed ist (mit flush), die URL umzuleiten führt zu einer *IllegalStateException*

3.3.4 Setzen der Cookies

Cookies wurden schon in der vorherigen Objective eingeführt. Das Setzen eines Cookies auf die Festplatte des Clienten funktioniert mit der Methode

- public void addCookie(Cookie cookie);

Um mehrere Cookies zu setzen wird die Methode einfach mehrmals aufgerufen

3.4 Lebenszyklus eines Servlets

Describe the purpose and event sequence of the servlet life cycle: (1) servlet class loading, (2) servlet instantiation, (3) call the init method, (4) call the service method, and (5) call destroy method.

Ein Client sendet eine Anrage nicht direkt an ein Servlet, sondern an einen Container (z.B TomCat) der die Servlets verwaltet. Der Lebenszyklus eines Servlets ist genau definiert.

3.4.1 Laden und Instantiierung eines Servlets

Es ist die Aufgabe das containers, eine Servlet Klasse zu finden, zu laden und zu instantiieren. Standardmässig wird ein Servlet erst beim Bedarf eingeladen. Dies hat die Anfrage dass die erste

Anfrage länger dauert. Es ist aber auch leicht konfigurierbar dass ein Servlet sofort beim Start des Containers eingeladen wird.

3.4.2 Initialisierung eines Servlets

Es können initiale Parameter aus einer Konfigurationsdatei (web.xml). Typisch für initiale Parameter sind z.B: Datenbank Connect Strings. Man kann die Datenbank wechseln ohne in den Code eingreifen zu müssen.

Beim Initialisieren wird folgende Methode der Klasse *HttpServlet* aufgerufen-

```
public void init(ServletConfig config) throws ServletException;
```

Ein *ServletConfig* speichert die Initialien Parameter in einer Paramterliste, auf die man dann dauerhaft zugreifen kann.

3.4.3 Anfragen an das Servlet weiterleiten

Ist ein Servlet eingeladen und initialisiert, so können Anfragen eines Clienten an das Servlet weitergeleitet werden. Für ein HttpServlet wird folgende Methode aufgerufen.

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Je nach Http Methode in der Anfrage, wird innerhalb der *service* Methode die doGet(), doPost() usw. Methode aufgerufen.

3.4.4 Servlet zerstören

Ein Servlet muss sich nicht dauernd im Container befinden. Ein Container kann ein Servlet das z.B. lange nicht mehr verwendet wurde zerstören. Bei Bedarf muss wieder ein neues Servlet eingeladen und neu instaniiert und initialisiert werden.

Vor dem zerstören eines Servlets wird die Methode *destroy()*aufgerufen. Damit hat das Servlet noch die Chance wichtige Daten zu speichern.

```
public void destroy();
```

Construct the file and directory structure of a Web Application that may contain (a) static content, (b) JSP pages, (c) servlet classes, (d) the deployment descriptor, (e) tag libraries, (f) JAR files, and (g) Java class files; and describe how to protect resource files from HTTP access.

Ein Container für Servlet und Java Server Pages wie TomCat, bietet dem Clienten i.a. mehrerer Web-Applikation an. Eine Webapplikation besteht aus eine vielzahl von Dateien, insbesondere aus Servlets, Jsps, Java-Klassen und Bibliotheken. Diese Dateien sind in einer fest vorgelegten Verzeichnisstruktur abgelegt. Kenntnisse über diese Struktur ist Inhalt dieser Objective

Eine typische Struktur sieht folgendermassen aus:

```
helloapp
       |
       |__  *.htm, *.css, *.xml
```

```
      |
      |__  *.jsp
      |
      |_ WEB-INF
          |
          |_classes
          |   |_com
          |      |_mycompany
          |         |_ myclass.class
          |
          |_ lib
          |    |_ *.jar (jdbcdriver.jar, mytaglib.jar etc.)
          |
          |_ tags
          |
          |
          |- web.xml
```

- Der oberste Ordner hat den Namen der Web Applikation
- Alle Dateien ausser des WEB-INF Ordners sind für den Clienten öffentlich zugänglich. Alle Dateien innerhalb des WEB-INF Ordners sind nicht für den Clienten zugänglich.
- Die Dateien ausserhalb von WEB-INF können in Ordner verschachtelt sein.
- Im Ordner WEB-INF sind alle Informationen die der Container braucht, und das Servlet zu starten. Es besteht mindestens aus dem Ordner classes, lib und der Datei web.xml
- In Classes sind alle Klassen die entsprechend der Paketstruktur in Verzeichnissen angelegt sind.
- in lib befinden sich alle jar Bibliotheken.
- Der Deployment-Deskriptor web.xml wird benutzt um die Web Applikation zu konfigurieren. Jede Web-Applikaton muss eine Datei web.xml enthalten.
- Zusätzlich können im Verzeichnis tags Tag Dateien enthalten sein.

Auf den folgenden Seiten wird die Struktur des Deployment Descriptors mittels Syntaxdiagramme aus der Servlet 2.4 Spezifikation von Sun dargestellt. Die folgende Legende zeigt, wie die Syntaxdiagramme zu lesen sind.

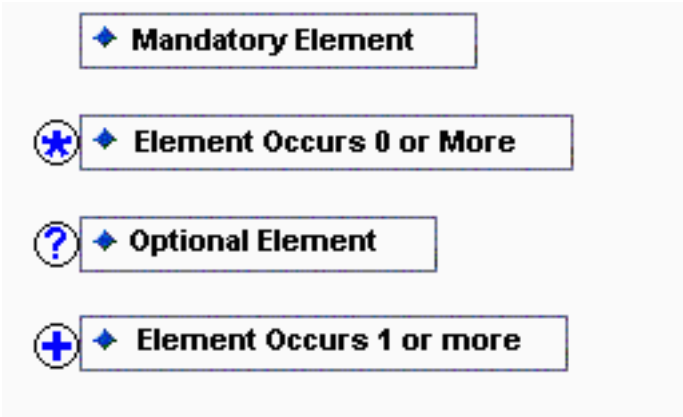


Abb. 3.1: Legende der Webelemente

Die Toplevel Elemente von web.xml werden durch das folgende Bild ersichtlich. Zum Glück sind nicht alle Elemente Gegenstand des Examens:

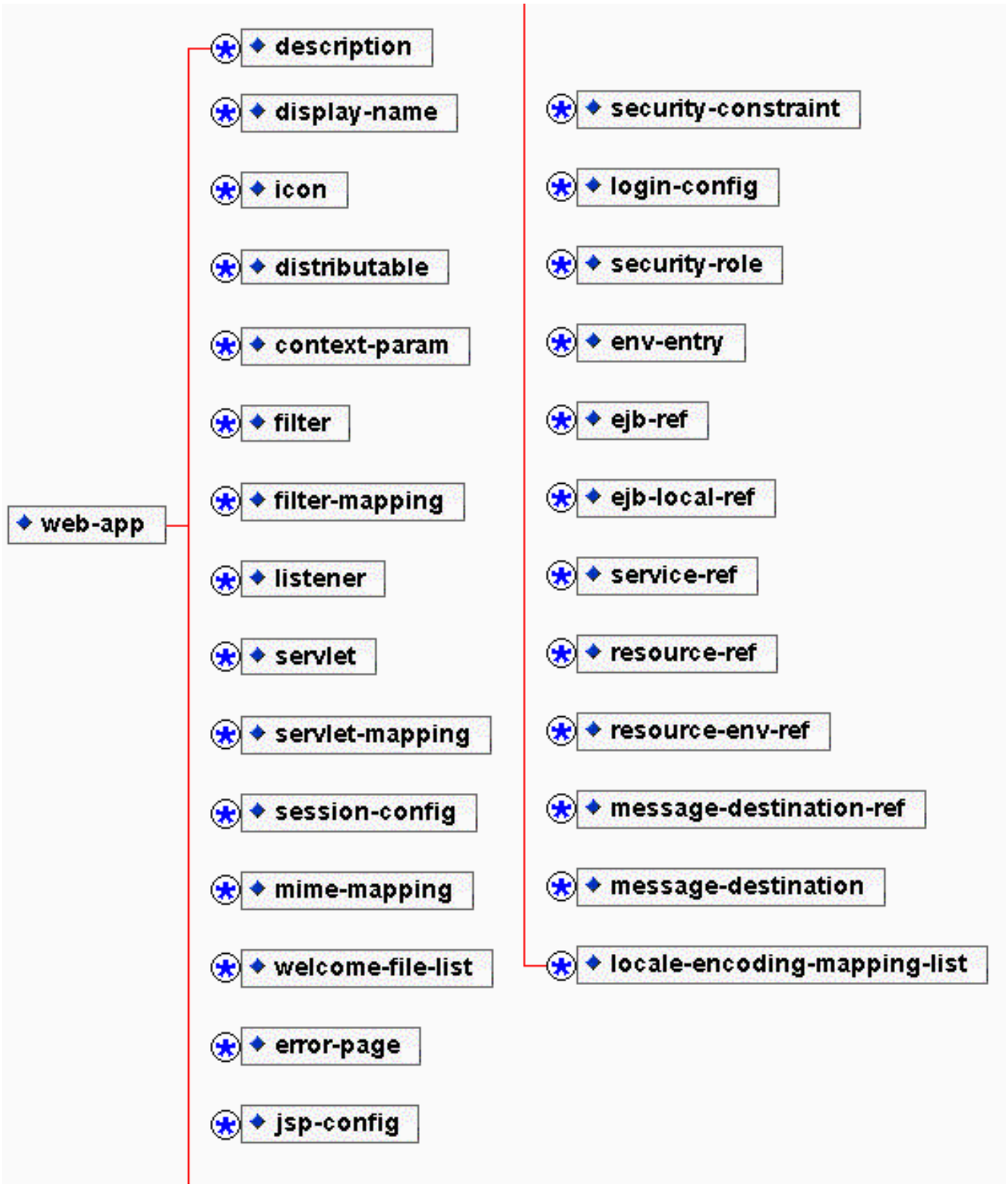
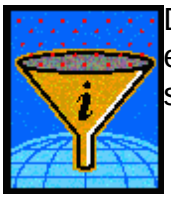


Abb. 3.2: TopLevel Elemente webApp

4 Semantik von web.xml



Describe the purpose and semantics for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

4.1 Error-Pages

Das *error-page* Element, definiert Seiten die den Anwender über aufgetretene Fehler informiert.

Die Syntax

```
<error-page>
    <error-code>errorCode</error-code> |
        <exception-type>className</exception-type>
    <location>pagePath</location>
</error-page>
```

Ein Fehler auf der Server-Seite kann durch eine Exception ausgelöst werden, oder durch die Anweisung *sendError(anInteger) bzw. sendError(anInteger, aString)*

Folgende Exception kann ein Servlet oder ein Filter auslösen:

- ServletException oder Unterklassen davon
- IO Exception oder Unterklassen davon

Treten andere Fehler auf, so können sie in ein ServletException gewrappt werden.

```
try()
   // Code der auf Datenbank zugreift
catch(SQLException e){
   throw new ServletException("SQL Exception",e);
}
```

Die standardmaessige Fehlerroutine wird nur aufgerufen, falls der Container in der *error-page* Deklaration keine entsprechende Location findet. Es wird durch die durch die URL spezifizierte Ressource aufgerufen.

Erhält der ServletContainer eine ServletException oder eine Unterklasse, so ermittelt er mit der Methode *ServletException.getRootCause* die gewrappte Exception und benutzt die *error-page* Dekleration um eine Fehlerseite zu finden und umzuleiten.

Location ist eine relative URL zur Wurzelverzeichnis der Webapplikation. Sie spezifiziert einer HTML-Seite, Servlet oder Java Servlet Page. *Location* muss mit einem Slash "/" beginnen.

Beispiel

```
<error-page>
   <error-code>404</error-code>
   <location>/errors/404.html</location>
</error-page>
<error-page>
   <exception-type>javax.servlet.ServletException</exception-type>
   <location>/errors/excepton.jsp</location>
</error-page>
```

Hat der Container eine Fehlerseite gefunden, so können über folgende Attribute in der Anfrage genauere Information ausgelesen werden:

- javax.servlet.error.status_code - Integer mit dem Code der von sendError mitgeben wurde.
- javax.servlet.error.exception_type - Class Exception die ausgelöst wurde
- javax.servlet.error.message - String der bei sendError bzw. der nicht aufgefangen Exception mitgegebem wurde
- javax.servlet.error.exception - Die Exception die den Fehler ausgelöst hat
- javax.servlet.error.request_uri - Die aktuelle URI der Fehlerseite
- javax.servlet.error.servlet_uri - URI der Seite die den Fehler ausgelöst hat

4.2 Init-Parameter

```
<init-param>
   <param-name>aKey</param-name>
   <param-value>aValue</param-value>
   [<description>ein beschreibender Kommentar</description>]
</init-param>
```

Syntax

Das *init-param* ist dem *servlet* untergeordnet, und kann dort beliebig oft vorkommen. Es wird verwendet um ein Servlet mit Parameter zu initialisieren. Diese Parameter werden nachdem instantiieren eines Servlets eingelesen.Innerhalb eines Servlets muss der *param-name* eindeutig sein.

Die Klasse *ServleConfig* enthält speichert diese initialen Werte und können mit folgenden Methoden ausgelesen werden:

- getInitParameterNames() - Gibt als Enumeration alle Parameternamen zurück, ggf. eine leere Enumeration
- getInitParameter(String) - Gibt als String den Wert des Parameters zurück, oder null

4.3 mime-mapping

Das *mime-mapping* Element definiert Zuordnungen von Daten zu Applikationen, die sie benötigen.

Syntax

```
<mime-mapping>
   <extension>fileExtension</extension>
   <mime-type>mimeType</mime-type>
</mime-mapping>
```

Die meisten Container haben standardmaessig Mappings für die gebräuchlichsten Dateiendungen (wie .htm , .gif,) definiert.

4.4 servlet

Die untere Abbildung zeigt den Aufbau des *servlet* Elements.

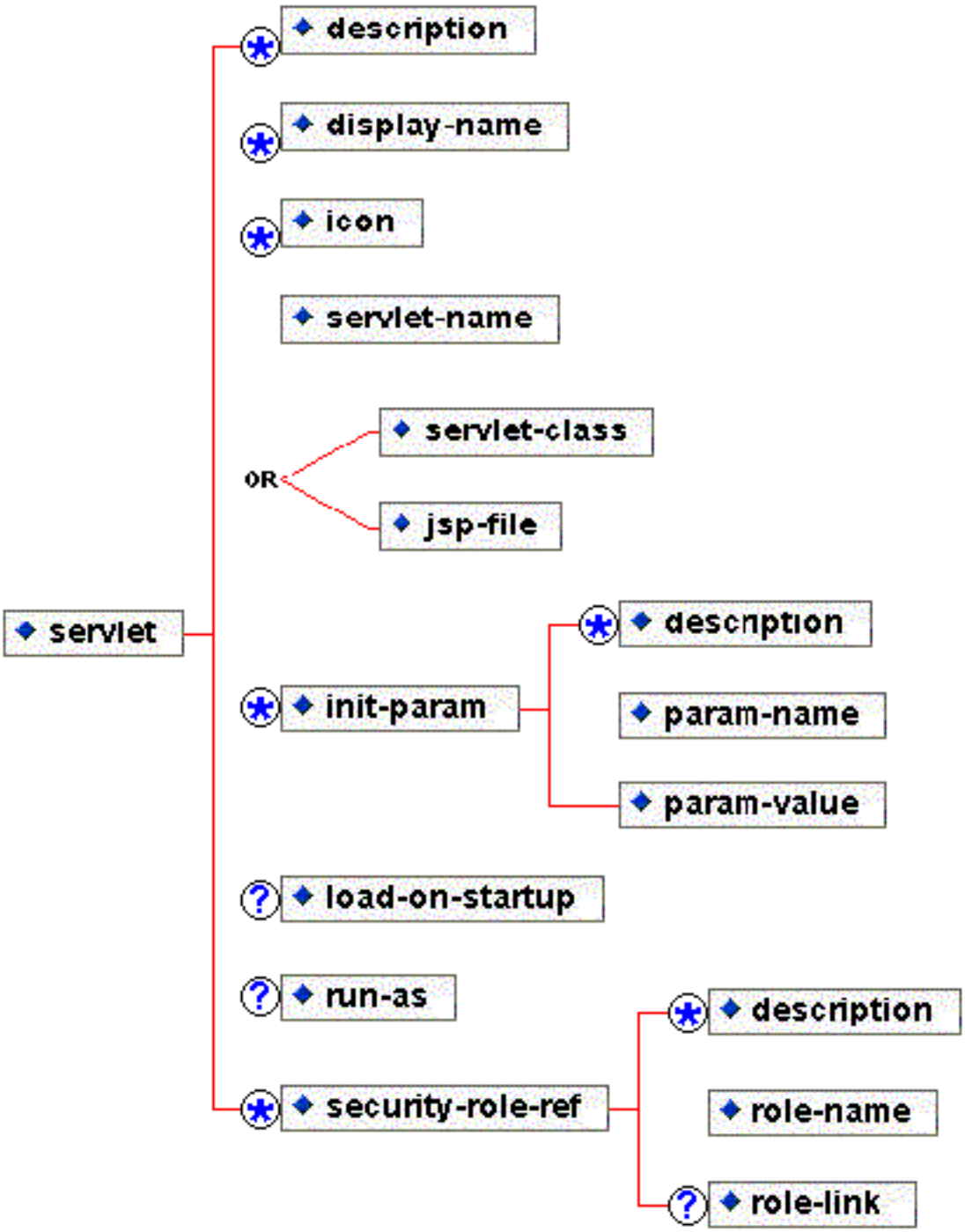


Abb. 4.1: Aufbau servlet Element in web.xml

Als Pflichtelement ist ein frei zu vergebender Name *servletName* und die entsprechende ServletKlasse *servlet-class* bzw. Java ServerPage Datei *jsp-page*. Der Servlet Name sollte eindeutig sein, dieslbe Servlet Klasse bzw. Java Server Page dürfen aber ruhig mehrfach vorkommen. Meistens sind dann die initialen Parameter *init-param* unterschiedlich. Es werden dann eben mehrere Instanzen einer Servlet Klasse erzeugt.

4.4.1 Icon, display-name und description

Diese Attribute sind identisch mit denen der TopLevel Elemente.

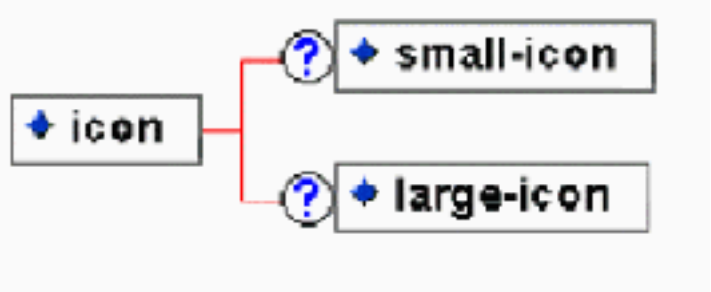


Abb. 4.2: Aufbau des Icon Elements in web.xml

Das Icon Element kann aus einem kleinen Icon (16*16 Pixel) oder grossen Icon (32*32 Pixel) bestehen. Der *display-name* bezeichnet den Namen der Applikation und *description* enthält eine längere Beschreibung. *Icon*,*Display* und *Description* können in mehreren Versionen vorkommen, unterscheiden sich aber dann in der Sprache die durch das Attribut *lang* eingestellt wird. Das Attribut *lang* hat als default Wert *"en"* für Englisch.

4.4.2 Init-Parameter und load-on-startup

Mit *init-param* werden Parameter mit denen die Servlets initialisiert werden spezifiziert und können durch die Klasse *ServletConfig* ausgelesen werden. Dies wurde auf der vorherigen Seite behandelt.

Bei einem positiven Wert von *load-on-startup* wird das Servlet gleich nach dem Hochfahren des Containers eingeladen und istantiiert. Anonsten wird das Servlet erst bei Bedarf instantiiert. Die Reihenfolge in der die Servlets eingeladen werden, hängt von der Höhe des mitgebenen Wertes ab.

4.4.3 run-as und security-role-ref

Rollen werden benutzt um Zugriffsberechtigungen von Benutzergruppen zu setzen. Rollen werden im Element *security-role* definiert. Die Zuordnung in *security-role-ref* dient dazu Rollennamen im Descriptor neu festzulegen, ohne den Code der Anwendung ändern zu müssen.

Hier ein Beispiel:

```
<security-role>
    Mitglieder der Controlling Abteilung
    <description> Mitglieder der Controlling Abteilung </description>
    <role-name>controller</role-name>
</security-role>

<security-role-ref>
    <description>Controller</description>
    <role-name>control</role-name>
    <role-link>controller</role-link>
</security-role-ref>
```

Der Sinn von *run-as* ist mir selbst noch unklar. Es wird benötigt wernn das Servlet einen Aufruf in einen EJB Container macht.

4.5 servlet-mapping

Das Element *servlet-mapping* mappt eine URL zu einem Servlet oder JSP.

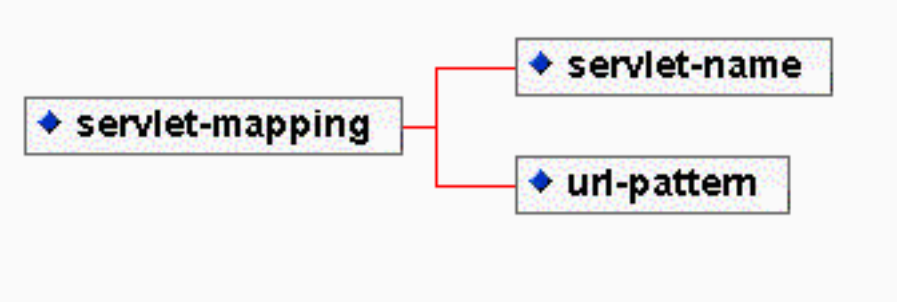


Abb. 4.3: Aufbau Element servlet-mapping in web.xml

Das Element *url-pattern* kann man in 4 Kategorien einteilen:

- Ein Prefix Pattern startet mit / und endet mit /*. z.B: */helloServlet/**
- Ein extension Mapping Pattern, startet mit *. z.B: **.pdf*
- Das Default Servlet Pattern besteht nur aus einem /
- Alles andere sind exakte Matches

Eine URL laesst sich auch aufspalten:

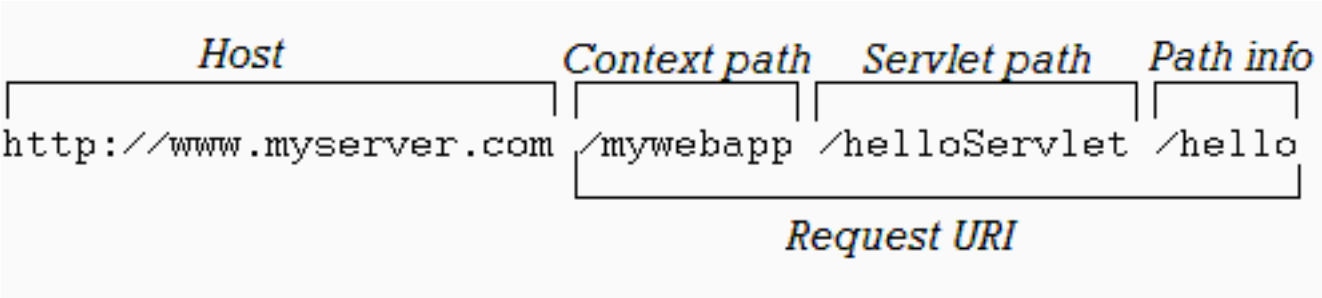


Abb. 4.4: Struktur einer URL

Beispiele:

```
<servlet-mapping>
   <servlet-name>purchase</servlet-name>
   <url-pattern>/po/*</url-pattern>
</servlet-mapping>
```

Ordnet die Urls dessen pathInfo mit /po/ anfaengt den Servlet purchase zu.

```
<servlet-mapping>
   <servlet-name>sales-report</servlet-name>
   <url-pattern>/report</url-pattern>
</servlet-mapping>
```

Beispiel fuer exaktes Matching pathInfo der URL muss /report sein.

```
<servlet-mapping>
   <servlet-name>XMLProcessor</servlet-name>
   <url-pattern>*.xml</url-pattern>
</servlet-mapping>
```

Eine URL mit PathInfo der mit *.xml endet wird dem Servlet XMLProcessor zugeordnet.

1. Der Container untersucht die servlet-mapping Elemente der in *contextPath* spezifizierten Webapplikation. Es wird nur nur der pathInfo Teil der URL untersucht.
2. Zunächst wird versucht ein Pattern mit exakten Matches zu finden.
3. Danach werden die *url-pattern* mit Präfix Pattern untersucht. Falls mehrere zutreffen, wird

das mit dem längsten Pfad genommen.
4. Falls immer noch kein Servlet gefunden wurde, werden die Mappings mit Extension Mapping Patterns untersucht.
5. Besteht der pathInfo Teil der URL lediglich aus einem Slash wird das default Servlet genommen.
6. Wurde immer noch kein Servlet gefunden, behandelt ein Default Prozessor Handler die Anfrage und es wird i.a. eine Fehlermeldung zurückgeben

4.6 welcome-file-list

Unter welcome File versteht man eine Datei die angezeigt wird, auch wenn in der URL nur ein Verzeichnis angegeben ist.

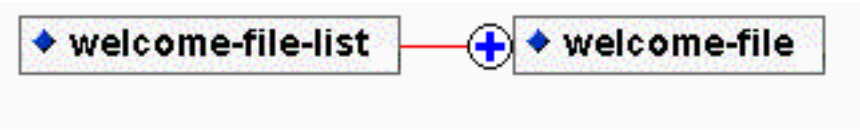


Abb. 4.5: Aufbau welcome file list

Beispiel:

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
</welcome-file-list>
```

Falls der Container mit Hilfe der servlet-mappings kein Servlet findet, versucht der Container durch Anfügen einer Datei in der welcome-file-list an die URL eine Ressource zu finden.

4.7 Struktur wichtigsten Elemente von web.xml

Construct the correct structure for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

Die Struktur der angegebenen Elemente wurde schon in den vorherigen Seiten definiert. Hier als Zusammenfassung die DTD's.

Error pages

```
<!ELEMENT error-page ((error-code | exception-type), location)>
```

Init parameters

```
<!ELEMENT init-param (param-name, param-value, description?)>
```

MIME mapping

```
<!ELEMENT mime-mapping (extension, mime-type)>

<!ELEMENT extension (#PCDATAT)>
```

```
<!ELEMENT mime-type (#PCDATAT)>
```

Servlet

```
<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
        (servlet-class | jsp-file), init-param*, load-on-startup?,
        security-role-ref*)>

<!ELEMENT servlet-name (#PCDATAT)>

<!ELEMENT servlet-class (#PCDATAT)>

<!ELEMENT jsp-file (#PCDATAT)>
```

Servlet mapping

```
<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

Welcome files

```
<!ELEMENT welcome-file-list (welcome-file+)>
```

4.8 Sinn, Aufbau und Erzeugung von war Dateien

Explain the purpose of a WAR file and describe the contents of a WAR file, how one may be constructed.

Komplexe Webanwendungen bestehen aus einem Bündel von zahlreichen Ressourcen z.B: JSP- und HTML Dateien, Medien Dateien, Java-Klassen, Konfigurationsdateien, Tag Bibliotheken oder Tag Dateien.

Um diese Dateien problemlos von einem Server zum anderen Verschieben zu können, müssen diese Dateien in einer definierten Verzeichnisstruktur abgespeichert werden. Alle Dateien für eine Webapplikation haben als Wurzel ein Verzeichnis mit dem Namen der Webapplikation.

Die Verzeichnisstruktur wurde schon erläutert hier nochmal die Zusammenfassung.

```
web_app   (bzw. Name der Webapplikation)
    öffentliche Dateien
    WEB-INF/
    WEB-INF/web.xml
    WEB-INF/classes/
    WEB-INF/lib/
    WEB-INF/tags
```

Es besteht die Möglichkeit, die komplette Struktur der Webanwendung in einem einzigen Archiv zusammenzufassen, das intern diese Struktur bewahrt. Dafür wurde ein spezielles Web Application Archive Format festgelegt, das mit der Dateierweiterung *.WAR* gekennzeichnet ist. Es ist ein normales JAR-Archive, nur mit einer anderen Extension. Diese Extension ist verbindlich, weil der Container daran, erkennt wie diese Datei zu behandeln ist.

Eine Webappliaktion mit dem Namen web_app wird folgendermassen archiviert

```
jar -cvf web_app.war *
```

5 The Web Container Model

5.1 Context Parameter

For the ServletContext initialization parameters: write servlet code to access initialization parameters; and create the deployment descriptor elements for declaring initialization parameters.

Wir haben gelernt, wie man innerhalb der Deployment Descriptor Datei *web.xml* mit dem Element *init-param* einem Servlet initiale Parameter mitgeben.

Die Servlets einer Web Applikation teilen sich ein Servlet Context. In der Datei *web.xml* lässt sich durch die Elemente *context-param* für ein Servlet Context initiale Paramter definieren.

```
<context-param>
    <param-name>webmaster</param-name>
    <param-value>myadress@mycompany.com</param-value>
    <description>The email adress the customer can write if he
    has installation problems
    </description>
</context-param>
```

Das Servlet Context Interface zeigt wie man auf diese Parameter zugreifen kann.

```
public interface ServletContext{
    public String getInitParameter(String name);
    public Enumeration getInitParamterNames();
}
```

5.2 scopes

For the fundamental servlet attribute scopes (request, session, and context): write servlet code to add, retrieve, and remove attributes; given a usage scenario, identify the proper scope for an attribute; and identify multi-threading issues associated with each scope.

5.2.1 Überblick

Innerhalb von Web-Applikationen kommunizieren verschiedene Servlets miteinander. Sie benötigen dazu einen gemeinsamen Bereich, mit dem sie Daten lesen bzw. austauschen können. Die Servlet Spezifikation definiert dazu 3 verschiededen Scopes, die es erlauben Attribute zu speichern und zu lesen.

Ihre Interfaces sind jeweils identisch, die Scopes unterscheiden sich lediglich in der Lebensdauer.

request	Gültig für eine einzige Anfrage (i.a. an ein Servlet). Ein Servlet kann mittels RequestDispatcher die Anfrage an mehreren anderen Servlets weiterleiten. Dann sind die Attribute, die in dem request gespeichert werden, auch in diesen anderen Servlets sichtbar. Wurde eine Anfrage abgearbeitet, so verliert man den Zugriff auf diese Attribute

session	Eine Session wird von einem einzigen Anwender in einem Browser angestossen, und ist solange gültig bis sich der Anwender explizit abmeldet, oder der Servlet Container die Session von sich aus beendet hat (z.B. falls lange keine Interaktion in dieser Session gemacht wurde). I.a. geht eine Session von einem einzigen Browser aus, es sei denn der Anwender öffnet aus diesem Browser aus einen 2. Fenster. (Wenn der Anwender dann noch Cookies verbietet funktioniert das Session Management nicht mehr)
context	Die Servlets einer Webapplikation teilen sich alle einen gemeinsamen ServletContext. Alle Anfragen aller Anwender dieser Webapplikationen können auf die Attribute des ServletContext zugreifen

5.2.2 Das Interface

Die folgende Tabelle zeigt, wie der Programmierer die verschiedenen Scopes erreicht.

request	Hier sind die Attribute an ein *(Http)ServletRequest* Instanz gebunden, die in der *doService()* Methode eines *HttpServlets* als Parameter mitgegeben wird
session	Von einer Instanz einer *HttpServletRequest* aus, bekommt man die aktuelle *HttpSession* durch die Methode *getSession()* bzw. *getSession(true)*
context	Zu dem Context eines Servlets gelangt man vom *HttpServlet* aus, durch die Methode *getServletContext()*

Das Interface um Attribute zu speichern, lesen und löschen sind für alle scopes identisch.

```
public interface ServletRequest (bzw. HttpSession oder ServletContext){
    Object getAttribute(String name);
    Enumeration getAttributeNames();
    void setAttribute(String name, Object o);
    void removeAttribute(String name);
}
```

Für die Attributnamen gibt es Konventionen:

- Attributnamen sollte man wie PackageNamen behandeln.Ein Attributname solle die Form haben wie z.B.: *meineFirma.meineWebApplikation.meinServlet.AttributName*
- Es sollte nie derselbe Attributname in verschiedene Scopes sein. Ggf können einige ServletContainer damit nicht umgehen.

5.2.3 Überlegungungen zum Umgang mit Threads

Ein WebServer wird i.a. paralell mit Anfragen bombadiert. Dabei kann es zu Problemen kommen, wenn ein Thread ein Attribut ändert, das gerade ein anderer Thread in Bearbeitung hat.

- Bei Request Attribute ist man auf der sicheren Seite, da wenn 2 Anwender eine Anfrage stellen, es verschiedene Instanzen dieses Servlets gebildet werden.
- Bei Session Attribute ist man leider nicht immer auf der sicheren Seite, da der Anwender 2 oder mehrere Fenster eines Browser geöffnet haben kann. Hier ist der Programmierer dafür verantwortlich, dass ein Thread nur exclusiv Zugriff auf ein Attribut bzw Session hat.

```
HttpSession session = req.getSession();
synchronized(session){
        session.removeAttribute(name);
}
```

Da ein Benutzer i.a. nicht hunderte von Browser geöffnet hat, ergibt sich dadurch aus Sicht der Performance keinen Flaschenhals.

- Context Attribute sind von allen Threads sichtbar und veränderbar. Natürlich kann man diese auch synchronisieren, allerdings kann man sich dadurch grosse Performance Probleme einhandeln. Man sollte deshalb im ContextScope nur Attribute benutzen, die sich sehr selten ändern.

5.2.4 Verhalten in einer verteilten Umgebung

Bei grösseren Anwendungen sind die Servlets auf mehreren Rechnern verteilt. So kann z.B. ein ServletContainer auf mehreren Maschinen mit verschiedenen Java Virtuellen Maschinen (JVM) arbeiten. Dies hat folgende Vorteile

- Fail-over support - Fällt eine Maschine aus, so kann einfach die andere Maschine, die Arbeit übernehmen.
- Load-balancing - ermöglicht eine weitgehend gleichmässige Auslastung der Server

Verteilte Anwendungen sind alles andere als trivial und wirken sich auf die Programmierung aus. D.h. Eine Applikation entwickelt für eine Maschine mit einer JVM kann nicht ohne groesseren Aufwand umkonfiguriert werden, so dass sie auf mehreren Maschinen läuft

- Wir können nicht annehmen, daß nur eine Instanz eines Servlets existiert, d.h. in einer verteilten Umgebung ist es problematisch Informationen in statischen bzw. Instanzvariablen zu speichern.
- Wir können nicht das lokale FileSystem benutzen.
- Verschiedene Maschinen haben verschiedene ServletContexte, d.h statt Attribute in einem ServletContext zu speichern, sollte man diese lieber in eine Datenbank abspeichern.

Verhalten von Servlet Context

Für jede Webapplikation gibt es pro virtuelle Maschine genau einen ServletContext. In einer verteilten Anwendung, gibt es aber i.a. mehrerer JVM und deshalb mehrere ServletContexte pro Webapplikation.

- Die Attribute eines ServletContext sind für einen anderen ServletContext nicht sichtbar. Es ist besser diese Attribute in einer Datenbank oder in einer Session zu speichern.
- Bei Änderungen in einem ServletContext können über Listener Objekte keine Methoden aus einem anderen JVM ausgelöst werden.
- Servlet Context Initialisierungs Parameter sind in allen JVM's sichtbar, da diese im Deployment Descriptor web.xml definiert worden sind

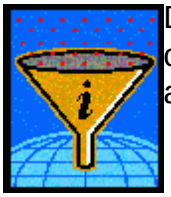
Verhalten von Http Session

Eine Session kann zu einem Zeitpunkt immer auf genau einer JVM laufen. Allerdings ist es erlaubt eine Session von einer JVM zu einer anderen zu migrieren.

- Eine HttpSession kann nie gleichzeitig auf 2 verschiedenen JVM laufen.
- Ein HttpSessionEvent kann nicht zu anderen JVM weitergeleitet werden.
- Damit eines Session auf eine andere Maschinen migriert werden kann, müssen ihre

Attribute das *Serializable* Interface erfüllen. Ansonsten wird eine *Illegal Argument Exception* ausgelöst.
- Wird eine Session migriert, werden alle Attribute die das *HttpSessionActivationListener* Interface erfüllen benachrichtigt.

5.3 Filter



Describe the Web container request processing model; write and configure a filter; create a request or response wrapper; and given a design problem, describe how to apply a filter or a wrapper.

5.3.1 Was sind Filter

Filter setzen sich vor einem Servlet. Sie können Anfragen abblocken, bzw ändern und sie können Antworten ändern. Mehrere Filter können zu einer Filterkette zusammengebaut werden.

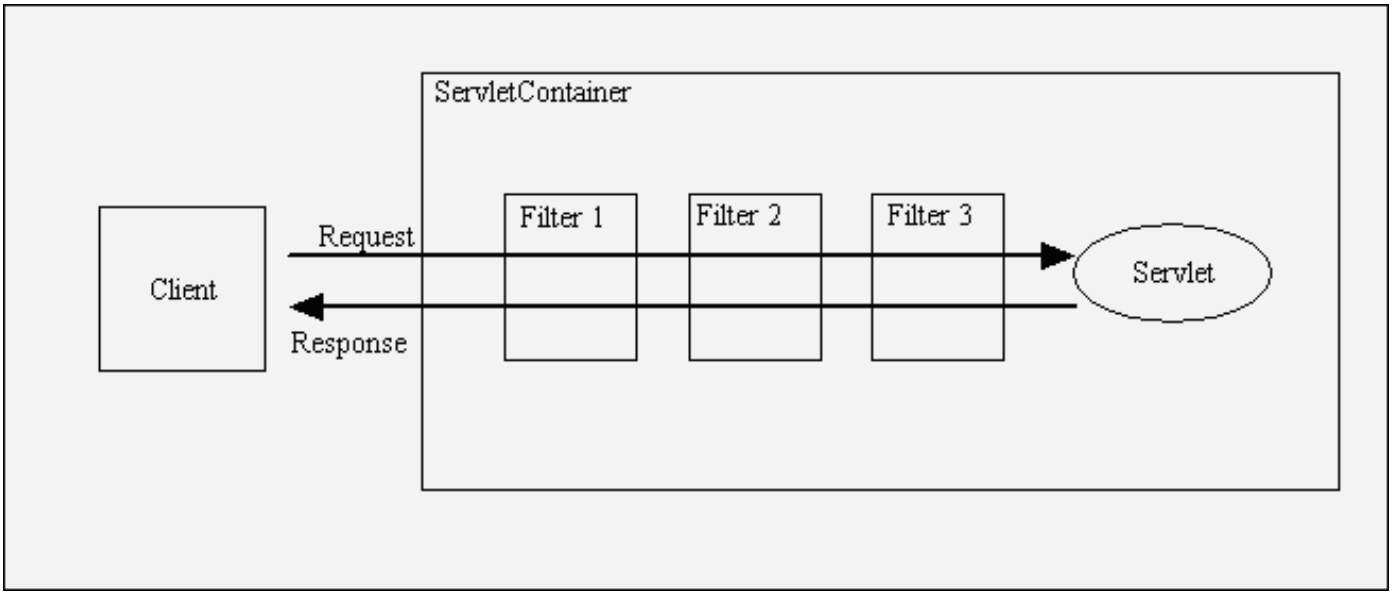


Abb. 5.1: Prinzip der Filter

Enthält ein Filter eine Anfrage, so hat er prinzipiell folgende Möglichkeiten:

- Er generiert die Antwort selber
- Er leitet die Anfrage (modifiziert oder nicht) zum nächsten Filter, oder falls er letzte Filter in der Kette ist an das Servlet
- Er leitet die Angabe an eine ganz andere Ressource um. D.h. der Filter nimmt die Rolle eines Dispatchers ein.

Typische Aufgaben eines Filters sind:

- Berechtigungsüberprüfungen
- Protokollierungen
- Anhang z.B: Bilder in ein Standardformat konvertieren.
- Daten komprimieren.
- Daten verschlüsseln.

- Triggern von Events

5.3.2 Das Request Processing Model

Wenn ein Servlet Container eine Anfrage erhält, passiert folgendes:

1. Anhand der *servlet-mapping* Elemente in Datei *web.xml* sucht er das Ziel-Servlet.
2. Anhand der *filter-mappging* Elemente in Datei *web.xml* sucht er die Filter die dem Ziel Servlet vorgeschaltet werden. Der Container laedt diese Filter entsprechend dem <filter> Elementen der Datei *web.xml*, instantiiert und initialisiert sie.
3. Der Container baut die gefundenen Filter zu einer Filterkette zusammen.
4. Die Anfrage wird an den ersten Filter geleitet.
5. Es ist dann die Aufgabe des jeweiligen Filters die Anfrage an den nächsten Filter weiterzuleiten (oder auch nicht).

5.3.3 Konfiguration einer Filterkette

Ein Filter wird in der Descriptor Datei nach folgender Syntax definiert

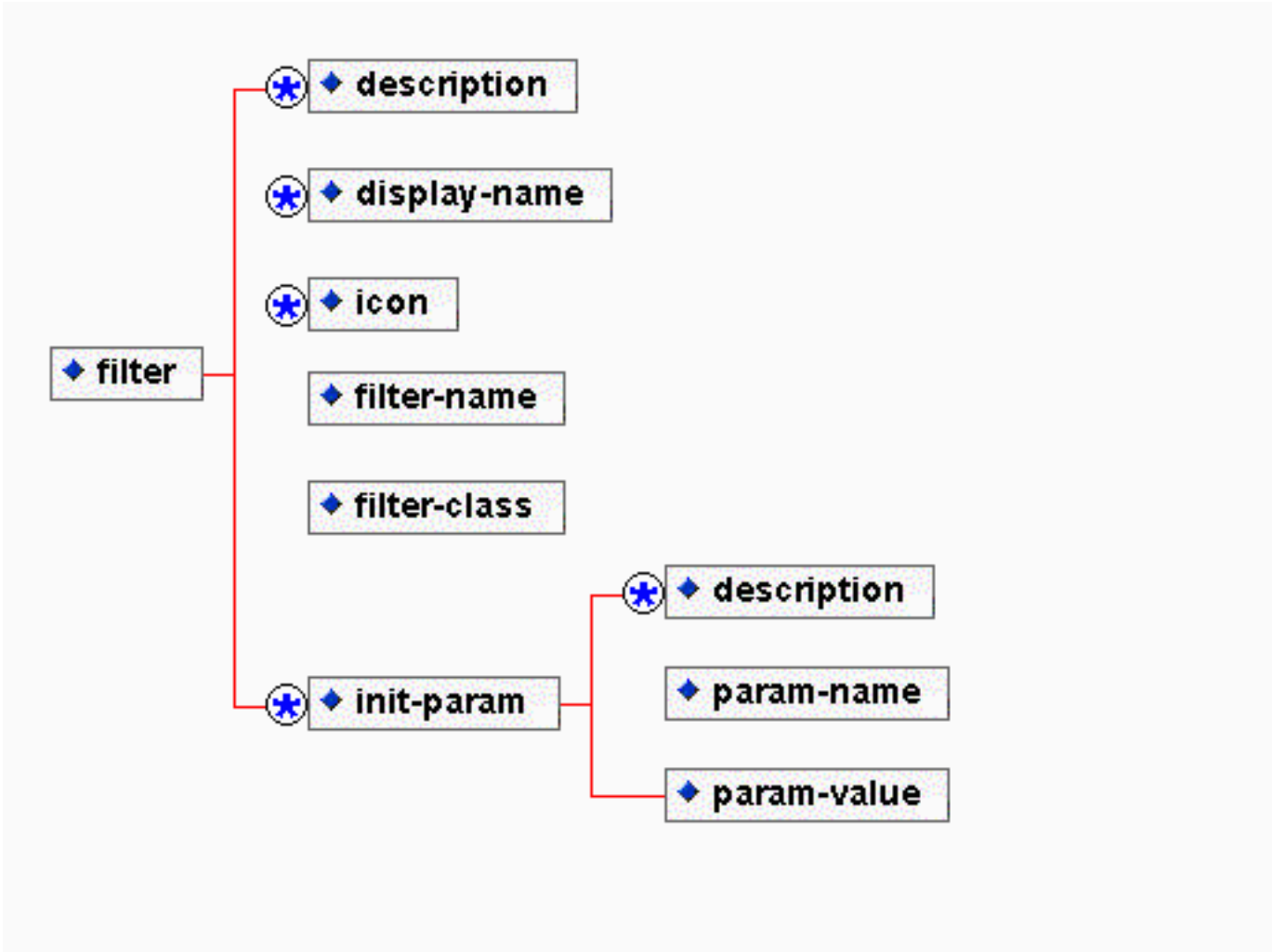


Abb. 5.2: Aufbau eines Filter Elementes

```
<filter>
        <filter-name>Validator Filter</filter-name>
        <description>Validates the requests</description>
        <filter-class>com.manning.filters.ValidatorFilter</filter-class>
        <init-param>
```

```
                <param-name>locale</param-name>

                <param-value>USA</param-value>
        </init-param>
</filter>
```

Ein *filter* Element hat fast den gleichen Aufbau eines *servlet* Elements, und braucht daher keine weitere Erklärung.

Entscheident welche Filter vor einem Servlet gestellt werden ist das *filter-mapping* Element.

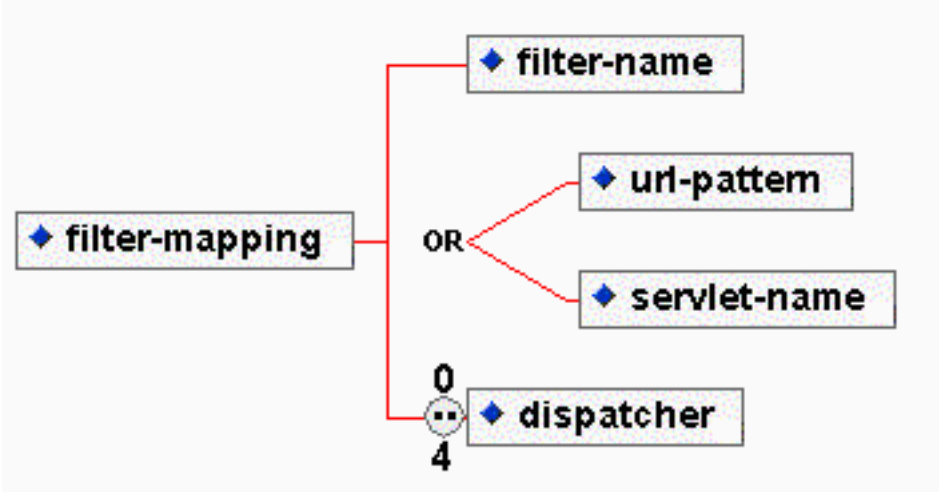


Abb. 5.3: Aufau eines Filter Mapping Elements

Beispiel:

```
<filter-mapping>
    <filter-name>ValidatorFilter</filter-name>
    <servlet-name>reportServlet</servlet-name>
</filter-mapping>

<filter-mapping>
    <filter-name>ValidatorFilter</filter-name>
    <url-pattern>*.doc</url-pattern>
</filter-mapping>
```

Das Element *dispatcher* kann innerhalb vom *filter-mapping* 4 mal vorkommen, und folgende Werte annehmen:

FORWARD	Filter wird angewendet, wenn die Anfrage auf das Servlet mittels *forward* Befehl und RequestDispatcher zugegriffen wird
INCLUDE	Filter wird angewendet, wenn die Anfrage auf das Servlet mittels *include* Befehl und RequestDispatcher zugegriffen wird
REQUEST	Dies ist die Standareinstellung und wird genommen, falls die Anfrage an das Servlet direkt vom Clienten kommt
ERROR	Dies wird genommen falls das Servlet ueber den *error-mapping* Mechanismus aufgerufen wird

Bemerkung: Der RequestDispatcher Mechanismus dient dazu Anfragen von einem Servlet an ein anderes einzubinden, ohne dass der Browser es mitbekommt (im Gegensatz zu sendRedirect()), Dieser Mechanismus wird später behandelt.

So findet ein Servlet Container die Filter zu einem Servlet:

1. Jeder Filter dessen url-pattern das Servlet matcht, wird als Filter genommen. Die Regeln für das *url-pattern* sind dieselbe wie beim *servlet-mapping* Element
2. Jeder Filter dessen *servlet-name* dem zugehörigen Servlet entspricht, wird als Filter genommen.
3. Die Reihenfolge der Filter wird folgendermassen definiert
   1. Nehme zuerst die Filter die durch ein *url-pattern* gematcht wurden.
   2. Nehme danach die Filter die durch *servlet-name* gematcht wurden.
   3. Innerhalb dieser 2 Gruppen entscheidet die Reihenfolge in der die Filter in der Datei web.xml definiert sind.

5.3.4 Das Filter Interface

Jeder Filter muss das folgende Interface erfüllen:

```
public interface Filter{
        public void init(FilterConfig filterConfig) throws ServletException;
        public void doFilter( ServletRequest request,
                              ServletResponse,
                              FilterChain chain) throws java.IOException, ServletException;
        public void destroy();
}
```

die Methoden init(), doFilter() und destroy() spiegeln den Lebenszyklus eines Filters wieder.

init() Methode

Die *init(FilterConfig)* Methode entspricht der *init(ServletConfig)* Methode eines Servlets. *FilterConfig* enthält alle Initialisierungsparameter aus der web.xml Datei.

Das folgende FilterConfig Interface zeigt, wie man diese Initialisierungsparameter ausliest.

```
public interface FilterConfig{
       public String getFilterName();
       public String getInitParameter(String name);
       public Enumeration getInitParameterNames();
       public ServletContext getServletContext();
}
```

doFilter() Methode

Die *doFilter()* Methode ist analog zur *service()* Methode eines Servlets. Beachte dass als Argument kein *HttpServletRequest* und *HttpServletResponse* besitzt, sondern ein *ServletRequest* bzw. *ServletResponse* übergeben wird. Im allgemeinen wird auf am Anfang einer *doFilter()* Methode, die Argumente zu den entsprechenden Http Typen gecastet.

Zusätzlich hat die doFilter() Methode ein FilterChain Objekt als Argument.

```
public interface FilterChain{
        public void doFilter ( ServletRequest request,
                       ServletResponse response )
                throws IOException, ServletException;
}
```

Das FilterChain Objekt dient dazu mit der *doFilter Methode* eine Anfrage an den nächsten Filter in der Filterkette (bzw. dem entgütigen Servlet) weiterzuleiten.

destroy() Methode

Die destroy() Methode wird als letzte Methode aufgerufen, kurz bevor der Filter zerstört wird. Sie ist analog der destroy() Methode eines Servlets.

5.3.5 Umgang mit ServletRequestWrappers bzw. -ResponseWrappers

Es existieren 4 verschiedene Wrapper Klassen:

- ServletRequestWrapper implements ServletRequest
- HttpServletRequestWrapper implements HttpServletRequest
- ServletResponseWrapper implements ServletResponse
- HttpServletRequestWrapper implements HttpServletResponse

Eine Wrapperklasse hat als Constructor genau ein Argument, das das Objekt enthaelt, was der Wrapper umwickelt. Dieses Argument hat ein je nach Art des Wrappers eine konkrete Klasse des (Http) ServletRequest bzw. (Http) ServletResponse.

Beispiel: Im unteren Beispiel, wird dem nächsten Filter kein HttpServletResponse Objekt mitgegeben, sondern einen Wrapper um das HttpServletResponse Objekt.

```
doFilter(ServletRequest req, ServletResponse resp, FilterChain chain){
       HttpServletResponse response = (HttpServletResponse) resp ;
       HttpServletResponseWrapper wrapper = new HttpServletResponseWrapper(response)
       chain.doFilter(resp, wrapper);
   }
```

Was haben wir dadurch gewonnen?. So wie das obige Beispiel implementiert ist, zunächst mal nichts. Der Wrapper hat dasselbe Interface wie das umwickelte Objekt. Die Anfragen an den Wrapper werden einfach an das umwickelte Objekt weitergeleitet.

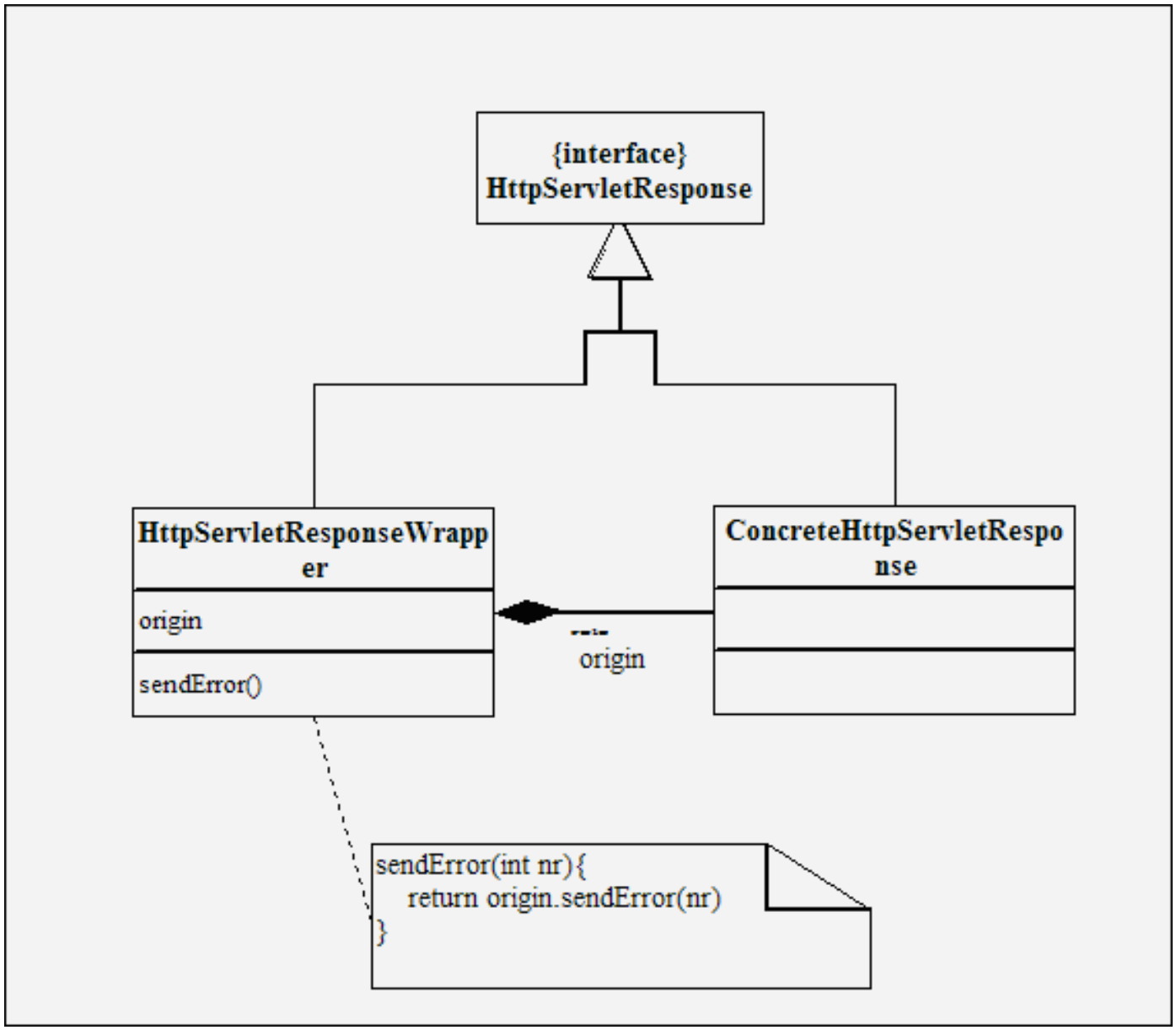


Abb. 5.4: Grundprinzip der Wrapper

Der Programmierer wird i.a. nicht direkt die angegebenen Wrapper benutzen, sondern Unterklassen davon verwenden.

```
doFilter(ServletRequest req, ServletResponse resp, FilterChain chain){
       HttpServletResponse response = (HttpServletResponse) resp ;
       HttpServletResponseWrapper wrapper = new UnterklassevonHttpServletResponseWrapper(response)
       chain.doFilter(resp, wrapper);
   }
```

In diese Unterklasse kann man z.B. implementieren dass die erzeugten Fehler mitprotokolliert werden.

Beispiel

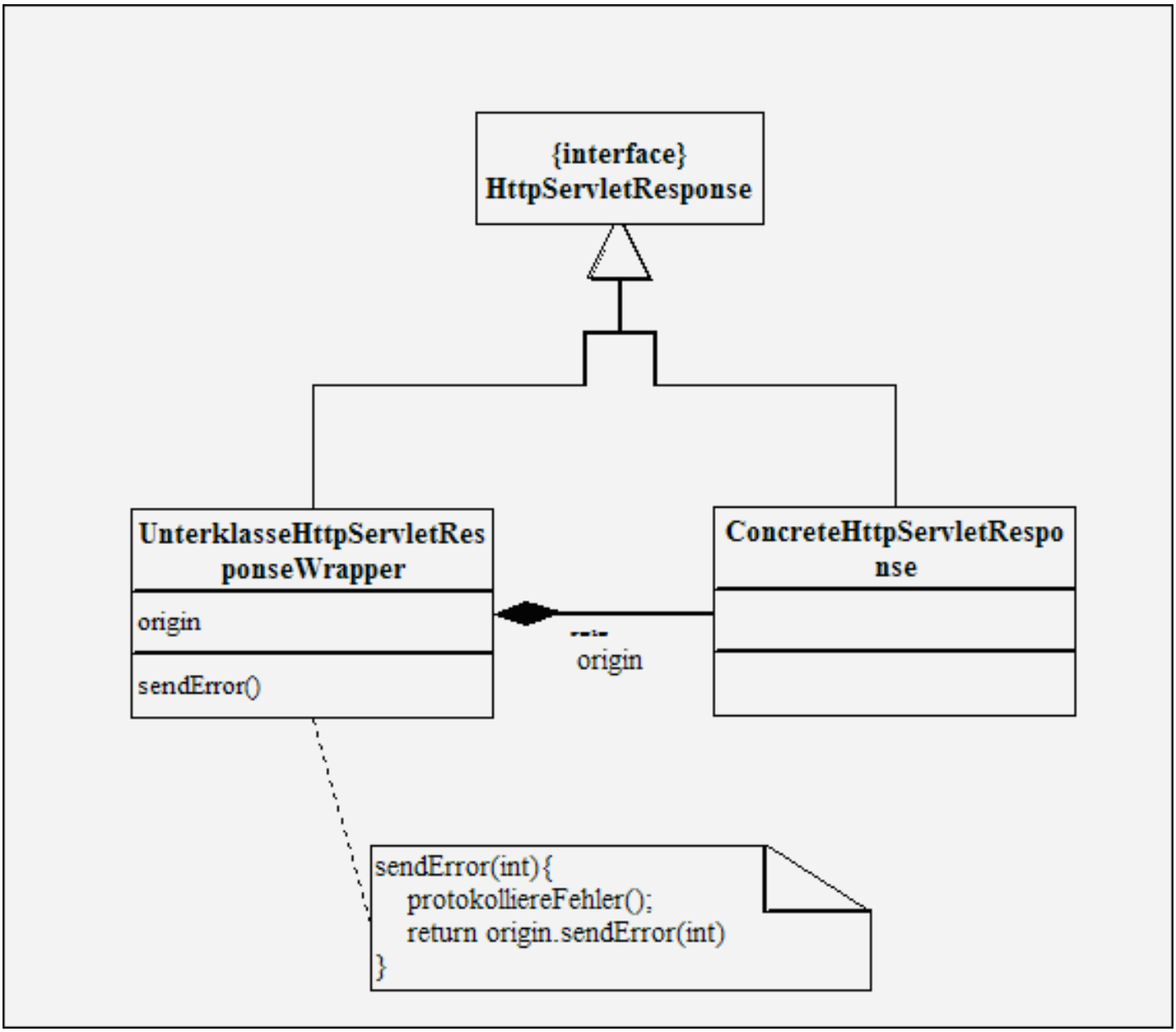


Abb. 5.5: Anwendungsbeispiel eines ResponseWrappers

Ein Wrapper dient dazu das Verhalten des umwickelten Objekts seinen eigenen Bedürfnissen anzupassen, ohne das umwickelte Objekt ändern zu müssen. Man könnte z.B. auch einen ResponseWrapper implementieren, der mit getOutputStream() statt einen normalen OutputStream ein Stream erzeugt, der die enthaltene Daten komprimiert.

Beachte HttpServletResponse ist ein Interface und keine Klasse. Der Wrapper weiss gar nicht, welche konkrete Klasse das Objekt besitzt, das er umwickelt. Der Wrapper kann aber trotzdem das Verhalten des umwickelten Objekts seinen Bedürfnissen anpassen.

5.4 Listener

Describe the Web container life cycle event model for requests, sessions, and web applications; create and configure listener classes for each scope life cycle; create and configure scope attribute listener classes; and given a scenario, identify the proper attribute listener to use.

5.4.1 Überblick der Listener Klassen

Instanzen von Listener Klassen beobachten Objekte. Wenn sich am Status dieser Objekte etwas ändert werden Events ausgelöst. Listener können auf diese Events reagieren.

Es können Listener Klassen definiert werden, die den ServletContext, eine Session, bzw. ein Request beobachten. Es werden Events ausgelöst wenn Attribute hinzugefügt, gelöscht oder geändert werden, bzw. wenn einer der Scopes initialisiert bzw zerstört wird bzw. eine Session migriert (bei verteilten Anwendungen kann die Session von einer JVM zu einer anderen verlagert werden).

Scope	**Event**	**Listener Interface**	**Event Klasse**
Servlet Context	Initialisierung und Zerstörung	ServletContextListener	ServletContextEvent
	Änderungen an Attributen	SevletContextAttributeListener	ServletContextAttributeEvent
HttpSession	Erzeugung und Zerstörung	HttpSessionListener	HttpSessionEvent
	Änderungen an Attributen	HttpSessionAttributeListener	HttpSessionBindingEvent
Servlet Request	Initialisierung und Zerstörung	ServletRequestListener	ServletRequestEvent
	Änderungen an Attributen	ServletRequestAttribute Listener	ServletRequestAttributeEvent

Zusätzlich gibt es noch die Listener **HttpSessionActivationListener** und **HttpSessionBindingListener**, die nicht die Session selbst sondern die Attribute innerhalb einer Session beobachten.

Scope	**Event**	**Listener Interface**	**Event Klasse**
Session - Attribut	Migration (Aktivierung und Deaktivierung)	HttpSessionActivationListener	HttpSessionEvent
	Hinzufügen oder Entfernen zu einer Session	HttpSessionBindingListener	HttpSessionBindingEvent

5.4.2 Listener die den Lebenzyklus eines Scopes beobachten

ServletContextListener

Direkt nachdem ein ServletContext instantiiert wurde und bevor die dazugehhörigen Servlets und

Filter geladen wurden, wird werden alle Listener, die das ServletContextListener Interface erfüllen benachrichtigt und die Methode contextInitialized(ServletContextEvent e) ausgeführt.

Wenn ein ServletContext zerstört wird, (geschieht i.a. nur beim Herunterfahren des ServletContainers) werden nachdem die ganzen Servlets und Filter innerhalb dieses ServletContext noch die Methode *contextDestroyed(ServletContextEvent e)* aufgerufen.

Das Interface hat folgende Methoden:

```
public interface ServletContextListener extends java.util.EventListener {
    public void contextDestroyed(ServletContextEvent sce);
    public void contextInitialized(ServletContextEvent sce);
```

Die Eventklasse ist folgendermassen aufgebaut:

```
public class ServletContextEvent extends java.util.EventObject {
        public ServletContext getServletContext();
}
```

HttpSessionListener

Ein HttpSessionListener reagiert darauf wenn eine Session erzeugt, bzw beendet wurde. Das Interface hat folgende Methoden:

```
public interface HttpSessionListener extends java.util.EventListener {
    public void sessionCreated(HttpSessionEvent e);
    public void sessionDestroyed(HttpSessionEvent e);
}

public class HttpSessionEvent extends java.util.EventObject {
        public HttpSession getSession();
}
```

ServletRequestListener

Eine Anfrage wird initialisiert bevor diese den ersten Filter vor dem Servlet betritt, und wird beendet, nachdem diese den letzten Filter verlässt.

```
public interface ServletRequestListener {
            requestInitialized(ServletRequestEvent e);
            requestDestroyed(ServletRequestEvent e);
}

public class ServletRequestEvent extends java.util.EventObject {
            public ServletRequest getServletRequest();
            public ServletRequest getServletContext();
}
```

Eintrag in web.xml

Die Klassen der Listener Objekte, müssen in der Datei web.xml unter dem listener Element eingetragen sein.

```
<web-app>
  <listener>
        <listener-class>myWebapp.myPackage.MySpecialServletContextListener</listener-class>
  </listener>
</web-app>
```

Die Listener werden beim Hochfahren des Containers in der Reihenfolge registriert, in der sie in der web.xml Datei aufgelistet sind. Beim Herunterfahren wird die umgekehrte Reihenfolge der Dekleration genommen, wobei die HttpSessionListener den ServletContextListener den Vorzug gegeben wird.

Der Servlet Container instantiiert selbständig die Listener Klassen. Die Listener Klassen dürfen deshalb kein Argument im Constructor besitzen.

Beispiel: Session Counter Listener

```
public class SessionCounterListener implements HttpSessionListener{
        private static final String COUNTER_ATTR = "session_counter";

        public void sessionCreated(HttpSessionEvent hse){
        Counter counter = getCounter(hse);
        counter.increment();
}

public void sessionDestroyed(HttpSessionEvent hse){
         Counter counter = getCounter(hse);
         counter.decrement();
}

private Counter getCounter(HttpSessionEvent hse){
    HttpSession session = hse.getSession();
    ServletContext context = session.getServletContext();
    return (Counter) context.getAttribute(COUNTER_ATTR);
}
```

5.4.3 Listener die Änderungen in der Attributliste ihres Scopes beobachten

Für jeden Scope gibt es Listener, die darauf reagieren, wenn ein neues Attribut in den Scope hinzugefügt wird, ein Attribut durch ein anderes Attribut ersetzt oder ein Attribut aus dem Scope gelöscht werden.

Je nach Scope heissen die Listener:

- ServletContextAttributeListener
- HttpSessionAttributeListener
- ServletRequestAttributeListener

Die Interfaces besitzen die Methoden: (Ersetze "scope" durch ServletContextAttribute bzw. HttpSessionBinding oder ServletRequestAttribute)

- public void attributeAdded("scope"Event e)
- public void attributeReplaced("scope"Event e)
- public void attributeRemoved("scope"Event e)

Die "scope" AttributeEvent Objekte kennen z.B folgende Methoden

```
public class ServletRequestAttributeEvent extends ServletRequestEvent {
        public String getName();
        public String getValue();
}
```

Auch verwendete Listener Klassen muessen im Deployment Descriptor web.xml bei den *Listener* Elemente registriert sein. Der Container instantiiert diese Klassen selbstsändig.

Vorsicht Falle

Beachte ein Listener bemerkt es nicht, wenn ein Attribut geändert wird.

Im unteren Beispiel wird kein Event getriggert, obwohl sich der Wert des Zaehlers ändert.

```
Counter counter = servletRequest.getAttribute("Zaehler");
     counter.increment():
```

Im folgenden Beispiel wird ein Event getriggert. Gibt es das Attribut "Zaehler" in dem RequestScope schon, so wird die Methode attributeReplaced(e) aufgerufen; ansonsten wird die Methode attributeAdded(e) aufgerufen

```
Counter counter = servletRequest.setAttribute("Zaehler",new Counter());
      counter.increment():
```

5.4.4 Listener für Session Attribute

HttpSessionActivationListener

In einer verteilten Umgebung kann eine Session von einer JVM zu einer ander JVM migrieren. Die Session, die auf der alten Viruellen Maschinen läuft, wird deaktiviert. Danach wird die Session auf der neuen JVM übertragen und instantiert, und diese Session danach aktiviert.

Mit Hilfe von *HttpSessionActivationListener* kann der Programmierer darauf reagieren.

```
public interface HttpSessionActivationListener extends java.util.EventListener {
    public void sessionDidActivate(HttpSessionEvent e);
    public void sessionWillPassivate(HttpSessionEvent e);
}

public class HttpSessionEvent extends EventObject {
        public HttpSession getSession();
}
```

HttpSessionBindingListener

Der *HttpSessionBindingListener* unterscheidet sich von den anderen *Listener* Klassen. Klassen, die das **HttpSessionBindingListener** entsprechen, sind i.a. Attribute einer HttpSession. Die andere Listener beobachten den entsprechenden Scope. Der **HttpSessionBindingListener** Interface ist beobachtet das Attribut einer Session selbst.

```
public interface HttpSessionBindingListener extends EventListener{

    public void valueBound(HttpSessionBindingEvent event);
    public void valueUnbound(HttpSessionBindingEvent event);
}
```

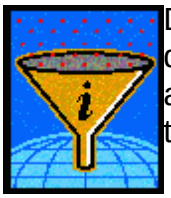
```
public class HttpSessionBindingEvent extends HttpSessionEvent {

    public HttpSession getSession ()
    public String getName() {
    public Object getValue() {
}
```

Alle Attribute die das *HttpSessionBindingListener* Interface erfüllen, werden benachrichtigt, falls das Attribut an die Session gebunden, oder von der Session gelöst wird. Die Benachrichtigung findet auch statt, falls die Session migriert, und das Attribut von der "alten" Session gelöst, bzw. an die "neue" Session gebunden wird.

Nicht jedes Attribut einer Session muss das *HttpSessionBindingListener* Interface erfüllen, auch gibt es für den ServletRequest und ServletContextScope keine entsprechenden Interfaces

5.5 Request Dispatcher



Describe the RequestDispatcher mechanism; write servlet code to create a request dispatcher; write servlet code to forward or include the target resource; and identify and describe the additional request-scoped attributes provided by the container to the target resource.

5.5.1 Erklärung Request Dispatcher

Die Aufgabe eines Request Dispatchers ist es eine Anfrage an eine andere Ressource (z.B ein Servlet, eine Html Seite oder JSP Datei) weiterzuleiten. Der Servlet Container

Ein RequestDispatcher wird durch eine der folgenden Methoden erzeugt

- aServletContext.getRequestDispatcher(String path)
- aServletContext.getRequestDispatcher(String name)
- aServletRequest.getRequestDispatcher(String path)

Die erste Methode *aServletContext.getRequestDispatcher(String path)* erwartet einen Pfad der mit einem Slash '/' beginnt. Der Pfad ist relativ zum Wurzelverzeichnis des *ServletContext*. Der Container versucht dann mit den *servlet-mapping* Regeln die Ressource zu finden. Es wird einen RequestDispatcher der das Zielobjekt umwrappt zurückgegeben, oder falls das Ziel Objekt nicht gefunden wird der *null* Wert zurückgegeben.

Mit der zweiten Methode *aServletContext.getRequestDispatcher(String name)* ist es möglich direkt auf eine Ressource zuzugreifen, wenn der Name dem ServletContainer bekannt ist. Auch hier wird ein RequestDispatcher oder der *null* Wert zurückgegeben.

Mit der dritten Methode *aServletRequest.getRequestDispatcher(String path)* kann man einen Pfad relative zu der aktuellen Servlet angeben.

5.5.2 Include und Forward

Das folgende Code-Fragment zeigt die Benutzung eines Request Dispatchers mit der forward Methode

```
public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        RequestDispatcher dispatcher =
             request.getRequestDispatcher(/template.jsp");
        if (dispatcher != null) dispatcher.include(request, response);
    }
}
```

Die forward Methode leitet die Anfrage an die template.jsp weiter. Mit der forward Methode ist es **nicht** möglich die Ausgabe der Qúellressource und der Ziel-Ressource zu mischen. Wurde schon vor der forward Anweisung etwas in den response Puffer geschrieben, geht dies verloren.Wurde der Puffer schon committed (ein *flush()* auf den entsprechenden OutputStream) erzeugt die forward Methode ein *IllegalStateException*.

Dasselbe Code-Fragment mit der include Methode:

```
public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        RequestDispatcher dispatcher =
             request.getRequestDispatcher(/template.jsp");
        if (dispatcher != null) dispatcher.include(request, response);
    }
}
```

Mit der Include Methode ist es durchaus möglich, die Ausgabe der Quell- und Zielressource zu mischen.

Beispiel:

```
public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        res.getContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("....");
        RequestDispatcher dispatcher =
             request.getRequestDispatcher(/template.jsp");
        if (dispatcher != null) dispatcher.include(request, response);
        out.println("..");
    }
}
```

Die Include Methode hat gegenüber der forward Methode einige Einschränkungen. Die Ziel Ressource kann nicht den Header von response verändern auch kann der Status Code von response nicht verändert werden.

5.5.3 Zusätzliche Attribute

Die Include Methode richtet automatisch in dem *HttpServlet* Objekt Attribute ein, die für den Fall dass das eingebundene Servlet oder die JSP-Seite diese Daten benötigen, den ursprünglichen Anfragepfad beschreiben. Diese Attribute, auf die die eingebundene Ressource durch Aufrufen von *getAttribute* auf dem *HttpServletRequest* zugreifen kann, sind nachfolgend aufgeführt. Diese Attribute müssen aber nicht gesetzt werden, falls der Include mit der *getNamedDispatcher* Methode ausgeführt wurde

- javax.servlet.include.request_uri
- javax.servlet.include.context_path
- javax.servlet.include.servlet_path
- javax.servlet.include.path_info
- javax.servlet.include.query_string

Dasselbe gilt auch für die forward Methode. Auch hier wird der Pfad der original Anfrage als Attribut automatisch eingebunden.

- javax.servlet.forward.request_uri
- javax.servlet.forward.context_path
- javax.servlet.forward.servlet_path
- javax.servlet.forward.path_info
- javax.servlet.forward.query_string

6 Session Manangement

6.1 Lebenszyklus einer Session

Write servlet code to store objects into a session object and retrieve objects from a session object.

Von einer Anfrage *HttpSessionRequest* erhält man eine Session durch eine der folgenden Methoden

- getSession()
- getSession(true)
- getSession(false)

Die Methode *getSession(false)* erzeugt keine neue Session, sondern gibt lediglich die aktuelle Session zurück. Existiert keine Session so wird *null* zurückgegeben.

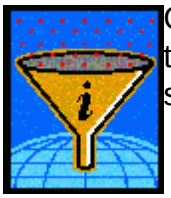
Die Methoden *getSession()* und *getSession(true)* verhalten sich identisch. Es gibt die aktuelle Session zurück. Existiert keine aktuelle Session wird eine neue erzeugt.

Das Interface um Objekte an eine Session zu binden, wurde schon bei den Scopes behandelt. Es ist nicht möglich für einen einzelnen Namen mehrere Objekte zu speichern.

Laut Konvention sollten die Namen AttributObjekte entspechend wie Packages aufgebaut sein. z.B com.myCompany.project.packageA.packageB.Object

```
public interface HttpSession {

        public Object getAttribute(String name);
        public Enumeration getAttributeNames();
        public removeAttribute(String name);
        public setAttribute(String name, Object value);
}
```



Given a scenario describe the APIs used to access the session object, explain when the session object was created, and describe the mechanisms used to destroy the session object, and when it was destroyed.

Die Methode *aHttpSession.isNew()* überprueft ob eine Session noch neu ist

- dem Clienten die SessionId gegeben wurde, und
- der Client die SessionId benutzt hat.

Eine Session wird beendet:

- Durch des ServletContainer, falls die Session eine gewisse Zeit inaktiv ist.
- Die Session wird explizit durch die Methode *aHttpSession.invalidate()* beendet.

Es ist die Aufgabe des Servlet Containers, die Sessions zu beobachten, zu messen wie lange diese inaktiv sind, und die inaktive Session zu beenden.

Man kann eine einzelnen Session einen Wert in Sekunden mitgeben, nachdem der ServletContainer bei Inaktivitaet diese Session löscht. Dies geschieht durch die Methode *setMaxInactiveInterval(int seconds)*. Ein negativer Wert setzt das Intervall auf unbegrenzt.

```
public interface HttpSession {

        public int getMaxInactiveInterval();
        public void setMaxInactiveInterval(int interval);
        public void invalidate();
        public boolean isNew();

}
```

Mit Hilfe des Deployment Descriptors kann man allen Sessions einer Webapplikation, einen Default Wert mitgeben, wie lange eine Session inaktiv sein darf, bis sie zerstört wird. Diese Angabe ist in Minuten (nicht in Sekunden wie bei *setMaxInactiveIntervla(int interval)*. Bei einem Wert gleich null oder negativ wird der Container eine Session nicht zerstören.

```
<web-app>
        ...
        <session-config>
                <session-timeout>30</session-timeout>  <!-- 30 minutes -->
        </session-config>
</web-app>
```

Ist das Element *session-config* nicht im Deployment Descriptor angegeben, wird ein interner Default Wert angenommen, der vom spezifischen Servlet-Container abhängt.

6.2 Session Listener

Using session listeners, write code to respond to an event when an object is added to a session, and write code to respond to an event when a session object migrates from one VM to another.

Es gibt zwei Listener Interfaces, die Objekte beobachten.

- Das Interface *HttpSessionAttributListener* beobachtet eine Session und es wird eine der Methoden *attributeAdded*, *attributeReplaced* bzw *attributeRemoved* ausgelöst, falls ein Objekt eine Session zugoerdnet, bzw entfernt wird. Hier ein Beispielcode:

```
public final class SessionListener implements HttpSessionAttributeListener {

  public void attributeAdded(HttpSessionBindingEvent event) {
                System.out.println("attributeAdded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + ", '" + event.getValue());
  }

  public void attributeRemoved(HttpSessionBindingEvent event) {
                System.out.println("attributeRemoved");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + ", '" + event.getValue());
   }

   public void attributeReplaced(HttpSessionBindingEvent event) {
                System.out.println("attributeReplaced");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + ", '" + event.getValue());
    }
}
```

Die Listener Klasse muss in der Datei web.xml deklariert werden

```
<web-app>
        ...
        <listener>
                <listener-class>packagesListener.SessionListener</listener-class>
        </listener>
</web-app>
```

- Mit dem *HttpSessionBindingListener* werden einzelne Attribute beobachtet. Dieses Interface wird nicht in der Datei web.xml deklarieren Hier ein Code Beispiel:

```
public class CounterBean implements HttpSessionBindingListener {

        private int count = 1;

        public int getValue() {
```

```
                return count;
        }

        public void increment() {
                count++;
        }

        public void valueBound(HttpSessionBindingEvent event) {
                System.out.println("attributeBounded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
        }

        public void valueUnbound(HttpSessionBindingEvent event) {
                System.out.println("attributeUnbounded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
        }
}
```

In einer verteilten Umgebung kann eine Session von einer JVM zu einer anderen migrieren. Intern wird die alte Session deaktiviert, die neue aktiviert. Die Attribute der Session müssen serialisiert und zur neuen Session migriert werden. Ein HttpSessionActivationListener beobachtet ein Attribut, und führt die Methoden *sessionWillPassivate()* bzw. *sessionWillActivate() aus* kurz vor oder direkt nach der Migration aus.

```
public class CounterBean implements HttpSessionActivationListener {

        private int count = 1;

        public int getValue() {
                return count;
        }

        public void increment() {
                count++;
        }

        public void sessionWillPassivate(HttpSessionEvent se) {
                System.out.println("session is about to be passivated");
                // save counter's value to persistent storage
                ....
        }

        public void sessionDidActivate(HttpSessionEvent se) {
                System.out.println("session has just been activated");
                // retrieve counter's value from persistent storage
                ....
        }
}
```

6.3 Techniken zum Session Management

Given a scenario, describe which session management mechanism the Web container could employ, how cookies might be used to manage sessions, how URL rewriting might be used to manage sessions, and write servlet code to perform URL rewriting.

Http ist ein zustandsloses Protokoll. Es ist ohne Tricks nicht möglich bei 2 verschiedenen Anfragen an den Server zu entscheiden, ob diese Anfragen vom selben Clienten kommen. Der Trick allgemein ist, bei der ersten Anfrage dem Clienten eine Sessionnummer zu geben, bei jeder weiteren Anfrage muss der Client diese Sessionnummer dem Server zeigen.

Es gibt folgende Möglichkeiten eine Sessionnummer zu vergeben, und weiterzuleiten.

- Cookie
- URL-Rewriting
- SSL Sessions

- speziell bei JSP ueber versteckte Formularfelder

Cookies

Cookies sind die einfachste und häufigste Methode, dem Clienten eine SessionId mitzugeben. Das Cookie muss den Namen **JSESSIONID** haben. Allerdings erlauben die meisten Browser dem Anwender das Speichern von Cookies abzuschalten.

SSL Sessions

Für den Austausch von sensiblen Daten sollte die Verschlüsselungstechnologie SSL (Secure Layer Socket) herangezogen werden, die von dem HTTPS-Protokoll verwendet wird. Sie bietet eingebaute Mechanismen, um eine Folge von Anfragen eindeutig einer Sitzung mit einem bestimmten Client zuzuordnen. Der Servlet-Container kann diese Daten verwenden, um eine Sitzung zu definieren und zu verwalten.

URL Rewrite

URL rewriting wird als Ersatz genommen, falls der Client Browser keine Cookies erlaubt. Die Idee ist einfach den sessionId an die URL anzuhängen. Der Name des Paramters heisst *jsessionid*. Beachte im Gegensatz zu den Cookies wird hier *jessionid* klein geschrieben.

Wenn man eine URL erzeugt muss garantiert sein, dass auch dort die SessionId angehängt wird. Dazu bietet das Interface in HttpServletResponse 2 Methoden:

- encodeURL(String)

Diese Methode gibt eine URL zurück, an der die sessionId angehängt ist.

- encodeRedirectURL(String)

Dies wird statt der sendRedirect() Methode genommen, wenn also die URL an eine andere Adresse umgeleitet wird.

Hier ein Code Beispiel:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        ...
        out.print("<form action='");
        out.print(response.encodeURL("SessionExample"));
        out.print("' ");
        out.println("method='post'>");
}
```

Im Zusammenhang mit Session Management sind noch solche Methoden interessant:

```
public interface HttpSession {

        public boolean isRequestedSessionIdFromCookie();
        public boolean isRequestedSessionIdFromURL();
        public boolean isRequestedSessionIdValid();

}
```

7 Web Application Security

7.1 Begriffsbestimmungen

Based on the servlet specification, compare and contrast the following security mechanisms: (a) authentication, (b) authorization, (c) data integrity, and (d) confidentiality.

Authentication

Darunter versteht man den Prozeß die Identität eines Clienten bzw. Anwenders festzustellen. Das einfachste Mittel ist durch Eingabe von Benutzernamen und Passwort. In Zukunft könnten das Scannen von Fingerabdruck, Augen Iris sein. Die Authentication beschäftigt sich nicht mit Rechtevergabe ist aber die Vorraussetzung dazu.

Authorization

Ist die Identität des Benutzers festgestellt, so ist der nächste logische Schritt die Authorization. Hier geht es darum festzustellen ob der Anwender die Rechte hat, auf die angeforderten Ressourcen zuzugreifen.

Data Integrity

Data Integrity ist der Prozeß zu versichern, dass während der Übertragung sich die Daten nicht verändert haben. Dies kann z.B. durch Prüfsummenbildung geschehen.

Confidentialy

Confidentialy sichert, dass kein unerlaubter Anwender vertrauliche Daten lesen kann. Während Authorization versucht den Zugang zu Daten zu regeln, versucht Confidentialy die Daten selbst dann zu schützen, falls ein Unbefugter die Daten schon bekommen hat. Dies geht durch Verschlüsselungsstrategien.

7.2 security constraint und login configuration

In the deployment descriptor, declare a security constraint, a Web resource, the transport guarantee, the login configuration, and a security role.

Eine Security Constraint definiert welche Ressourcen auf welche Art geschützt werden sollen. Sie besteht aus i.a. drei Unterelementen

- web-resource-collection - Enthält Liste von schützenswerten Ressourcen
- auth-constraint - Besagt wer darauf zugreifen kann
- user-data-constraint - Spezifiziert auf welche Art und Weise Ressourcen zum Clienten gesendet wird.

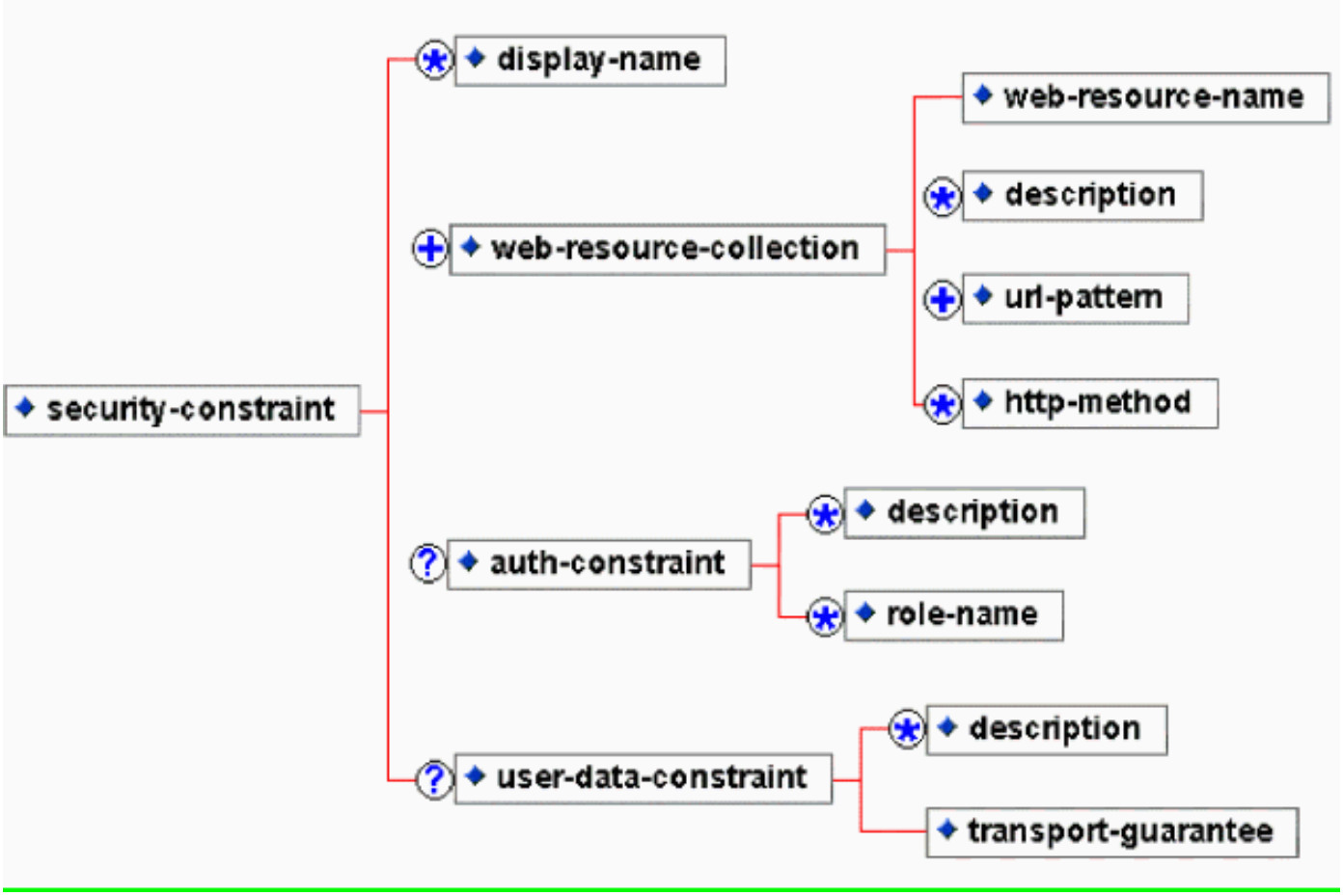


Abb. 7.1: Aufbau einer SecurityConstraint

web-resource-collection

Mit einem URL-Pattern spezifiziert man, welche Ressourcen zu schützen sind. Zusätzlich können noch Angaben zu den für den Zugriff auf die jeweiligen Ressource erlaubten HTTP-Methoden gemacht werden.

auth-constraint

Alle Benutzer, die die angegebene Rolle innehaben, können auf die geschätzten Web-Ressource zugreifen. Die Rollen müssen in der Datei web.xml in dem Element *security-role* definiert sein.

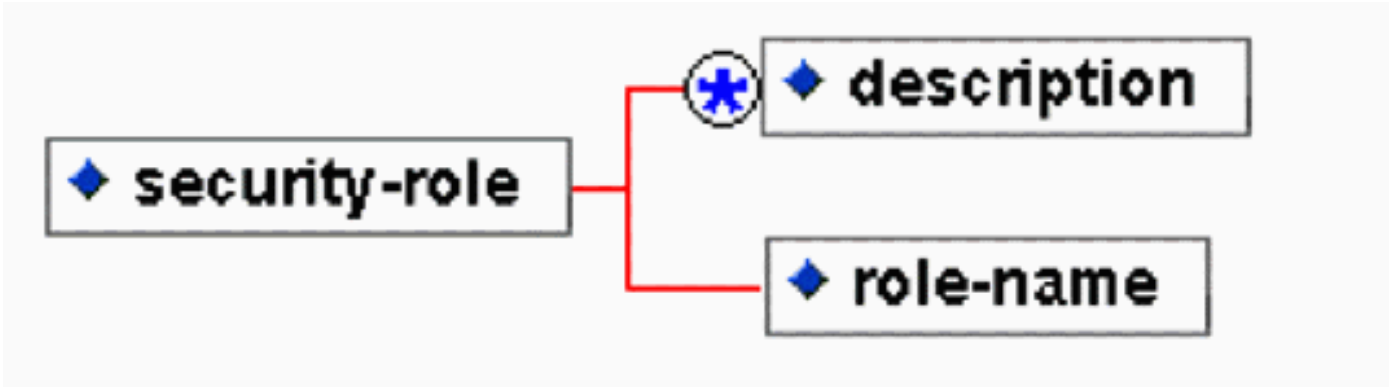


Abb. 7.2: Security Role

Beispiel:

```
<security-role>
    <description>
        Mitglied der Controlling-Abteilung
    </description>
    <role-name>controller</role-name>
</security-role>
```

Statt eines bestimmten Rollennamens kann auch der Matchcode '*' verwendet werden, soll die Ressource für alle der Anwendung bekannten Rollen freigegeben werden. Wird dagegen keine der Rollen angegeben, ist dieser Teil der Wbanwendung für alle Benutzer gesperrt.

user-data-constraint

Das Unterelement *transport-guarantee* kann folgende Werte annehmen:

Name	**Bedeutung**
None	Keine Anforderung (Default Wert)
Integral	Daten dürfen beim Versenden nicht verändert werden
Confidential	Daten dürfen beim Versenden nicht von Dritten eingesehen werden

Beispiel:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>admin</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Dieses security Constraint schützt alle Ressourcen deren UI mit /admin anfängt. Es dürfen nur Benutzer mit der Rolle admin auf diese Ressourcen zugreifen. Ausserdem kann der Zugriff nur über die get Methode erfolgen. Bei der Übertragen der Daten muss garantiert sein, dass diese nicht von Dritten einsehbar ist.

login-config

Mit dem Element *login-config* wird in der Datei web.xml bestimmt welche Typ zur Authentifizierung genommen wird (siehe weiter unter).

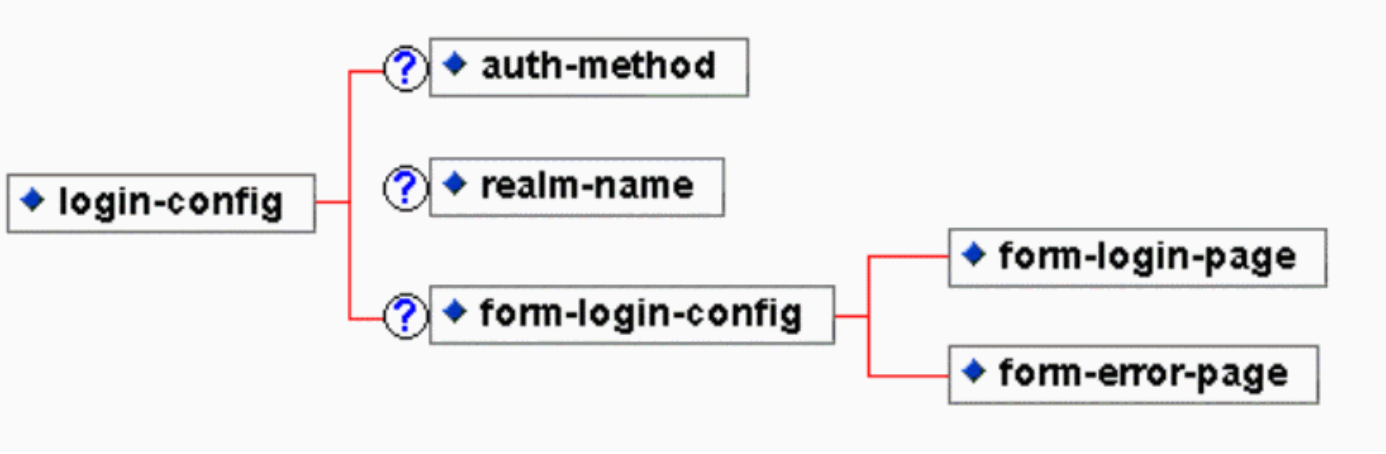


Abb. 7.3: Login-config

Beispiel:

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Protected pages</realm-name>
</login-config>
```

Das Element *auth-method* kann eine der Methoden BASIC,DIGEST,FORM oder CLIENT-CERT annehmen. Der realm-name wird bei dem Formalar das der Brwoser zur Eingabe von Benutzer und Passwort eingibt, angezeigt.

Wird die Authentifizierung FORM benutzt, muss die Seite zur Login Eingabe und die Fehlerseite falls der Login fehlerhaft ist angegeben werden.

```
<login-config>
    <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login/login.html</form-login-page>
            <form-error-page>/login/error.html</form-error-page>
        </form-login-config>
</login-config>
```

7.3 Authentifizierungs Typen

Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and given a scenario, select an appropriate type.

Es werden 5 Möglichkeiten angeboten, wie man einen Anwender indentifizieren kann.

Name	Bedeutung
BASIC	HTTP-Authentifizierung auf der Basis von Benutzername und Passwort
DIGEST	Verschlüsselte Übergabe des Passworts
FORM	Basis Authentifizierung, über ein benutzerdefiniertes Login-Formular
CLIENT-CERT	Https-Client-Authentifizierung

HTTP BASIC Authentication

Die Identitifizierung läuft folgendermassen ab.

- Der Anwender sendet eine Anfrage.
- Der Servlet Container erkennt dass die angeforderte Ressource geschützt ist.
- Der Servlet Container sendet eine Antwort zurück, diese Antwort hat folgenden Inhalt
  - StatusCode 401 UNAUTHORIZED
  - Es wird mitgegeben welche Authentication Methode benötigt wird. in diesem Fall BASIC
  - Es wird mitgegeben unter welchem Context(realm) die Authentication erfolgt.
- Der Browser des Anwenders liest die Antwort, und generiert einen Dialog. Der Dialog enthält
  - den Context String realm
  - Ein Eingabefeld für den Benutzernamen
  - Ein Eingabefeld für Passwort
  - Buttons zur Bestätigung der Eingaben bzw. Abbruch.
- Der Anwender gibt Benutzernamen und Passwort ein.
- Die ursprüngliche Anfrage wird nochmals gesendet, allerdings ist nun Benutzernamen und Passwort in den Header der Anfrage eingebaut.
- Der ServletContainer empfaengt die Anfrage, überprueft ob Passwort und Benutzername korrekt ist.
- Ist Passwort und Benutzername nicht korrekt, so gibt er wieder eine Antwort mit StatusCode 401 UNAUTHORIZED mit. Die beschriebene Prozedur wiederholt sich.
- Ist Passwort und Benutzername korrekt, so erhält der Anwender die Ressource

HTTP DIGEST Authentication

Die Identifizierung läuft fast identisch mit der Methode BASIC ab. Allerdings gibt es einen Mechanismus um das Passwort zu verschlüsseln.

HTTP FORM Authentication

Dies ist die einzigste Typ indem der Programmierer die Anmeldeseite selber gestalten kann. Folgende Konventionen gelten für eine selber formulierte Anmeldeseite:

- Es benötigt 2 Eingabefelder j_password und J_username
- Es benötigt ein Action Attribut j_security_check

HTTPS CLIENT-CERT Authentication

Es wird das HTTPS Protokoll (SSL-Verbindung) verwendet und die Daten werden mit einem Public-Key Verfahren verschlüsselt. Dies ist die sicherste Methode benötigt aber eine Zertifizierung von einer autorisierten Stellen. Genaue Kenntnisse über das Verfahren, ist für die Zertifizierung wahrscheinlich nicht notwendig.

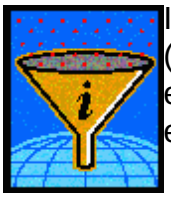
Die Vor- und Nachteile habe ich in einer Tabelle zusammengefasst

TYP	**VORTEILE**	**NACHTEILE**
BASIC	Jeder Browser unterstützt es	Aussehen der Login-Page nicht wählbar
	Sehr einfach zu konfigurieren	Sehr unsicher Passwort und Benutzername nicht

		verschlüsselt
DIGEST	Sehr einfach zu konfigurieren	Aussehen der Login-Page nicht wählbar
	Sicherer als BASIC. Es wird Passwort verschlüsselt	Nicht jeder Browser unterstützt es
FORM	Alle Browser unterstützen es	Nicht sicher da Benutzer und Passwort nicht verschlüsselt
	Look and Feel Login-Page wählbar	Sollte nur im Zusammenhang mit HTTPS oder Cookies verwendet werden
	Einfach zu konfigurieren	
HTTPS CLIENT CERT	Das sichertse aller Verfahren	Zertifizierung durch autorisierte Stelle notwendig
	Wird von allen Browser unterstützt	teuer

8 JavaServer Pages Technology Model

8.1 JSP Elemente



Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.

8.1.1 template Text

Alles was kein JSP Element ist ein Template Text. Templates können jede Art von Text sein: HTML,WML, XML oder einfacher Text. Auch wenn JSP fast immer im Zusammenhang mit HTML gebraucht wird, basiert JSP nicht auf HTML. Der Template Text wird so wie er ist, dem Browser mitgegeben. Der Template Text ist immer der statische Teil einer JSP, er wird mit den dynamischen Teil (den JSP Elementen) gemischt und das Ergebnis wird als Response Stream weitergereicht.

Möchte man ein JSP Element darstellen, ohne dass es interpretiert wird, so geht dies durch durch quoten

- A literal **<%** wird durch einen Slash nach der eckigen Klammer gequotet: <\%
- Expression Language können durch den einen Slash vor dem Dollarzeichen gequotet werden **\$**. Dies funktioniert aber nur, falls für die Seite die Expression Language eingestellt ist.

8.1.2 Skripting Elemente

Kommentare

Kommentare dienen nur der Dokumentation und werden folgendermassen gebildet

```
<%-- Dies ist ein Kommentar --%>
```

JSP Kommentare können nicht ineinander verschachtelt werden.

Dieser Kommentar wird nicht in die HTML generierte Seite eingebaut. Möchte man in die HTML generierte Seite einen Kommentar einbauen, so kann man dies als Template Text machen.

<!-- Dies ist ein Html Kommentar -->

Direktive

Die JSP Maschine übersetzt eine JSP Seite in ein Servlet. Mit den Direktiven kann man der JSP Maschine Anweisungen geben.

Es gibt 3 Arten von Direktiven:

- page - Definiert Eigenschaften der aktuellen JSP

```
<%@ page language="java" %>
```

Dies sagt der JSP Maschine, dass Java als Skriping Elemente eingebunden wird.

- include - Es wird der Inhalt einer anderen Seite in die aktuelle JSP eingebunden.

```
<%@ include file="copyright.html" %>
```

An dieser Stelle wird die Datei copyright.html eingebunden.

- taglib - Assoziiert ein Prefix mit einer TagLib Bibliothek

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

Zusammenfassend lässt sich sagen:

- Eine Direktive beginnt immer mit <%@ und endet mit %>
- Eine Direktive hat folgende Syntax

```
<%@ page  attribut-liste %> oder
<%@ include  attribut-liste %> oder
<%@ taglib  attribut-liste %>
```

Deklarationen

Deklarationen ermöglichen es Java Variablen oder Methoden zu definieren, und zu initialisieren Deklarationen beginnen mit <%! und endet mit %>. Es kann jede mögliche Java Code Definition beinhalten.

```
<%! String color[] = {"red", "green", "blue"}; %>

<%!
    String getColor(int i)
    {
        return color(i%3);
    }
%>
```

Beachte dass in Java jedes Statement mit einem Semikolon abgeschlossen wird. Ansonssten

weir ein Fehler eingebunden.

Skriplets

Skriplets sind Java Code Fragmente die in JSP Seiten eingebunden werden.

```
<% variable++; %>
```

Das Skriplet wird jeweils ausgeführt, sobald eine Anfrage an das Servlet erfolgt. Ein Skriplet beginnt mit <% und endet mit %>. Dazwischen stehen ein oder mehrere Java-Statements. Beachte dass die Anweisungen mit einem Semikolon abgeschlossen werden müssen.

Expressions

Expressions sind Java Ausdrücke, deren Auswertung in die Ausgabe eingebunden werden. Beispiele:

```
<%= userInfo.getUserName() %>
<%= 1 + 1 %>
<%= new java.util.Date() %>
```

Es ist zu beachten das im Gegensatz zu den Skriplets, Ausdrücke nicht mit einem Semikolon abgeschlossen werden. Ausdrücke beginnen mit <%= und enden mit %>.

Hier einige Beispiele die keine Ausdrücke sind:

<%= aBool; %>	Ein Ausdruck wird nicht mit einem Semikolon beendet
<%= int i=20 %>	Man kann in einem Ausdruck keine Variable oder Methode deklarieren
<%= sBuff.setLength(12) %>	Die Methode gibt keinen Wert zurück

Die Auswerung eines Ausdrucks muss kein String sein, aber in einen String konvertiert werden können.

Erkennt die JSP Maschine dies nicht schon zur Übersetzungszeit, so wird bei der Auswertung der Anfrage

ein ClassCastException ausgelöst.

8.1.3 Standard und Custom Actions

Aktionen sind Anweisungen an die JSP Maschine, die zur Laufzeit ausgeführt werden. Es gibt zwei Arten von Aktionen.

Zum einen dijenigen mit dem Prefix jsp, z.B: <jsp:include page="neAndereJsp.jsp" />; Dies sind die sogenannten Standard Actions. Custom Actions haben einen anderen Prefix als jsp.

Im Unterschied zu den anderen JSP-Elementen wie Direktiven und Skriptelemente wird für Aktionen nur noch die XML-Syntax genommen.

```
<prefix:aktionsname attributname1="wert1"
     attributname2="wert2" ...>

 Rumpf der Aktion
```

```
</prefix:aktionsname>
```

Innerhalb des Rumpfes können beispielweise Kindelemente, die Paramter an die Aktion übergeben, enthalten sein, oder auch Template-Daten, auf deren Ausgabe die Aktion in irgendeiner Form Einfluss nimmt.

Beispiel mit Paramterübergabe:

```
<jsp:include page="anbieterListe.jsp" >
   <jsp:param name="gebiet" value="Berlin" />
   <jsp:param name="status" value="A1" />
</jsp:include>
```

Kommt die Aktion ohne Rumpf aus so kann die Syntax auf

```
<prefix:aktionsname attributname1="wert1" />
```

verkürzt werden.

Standard Custom Actions müsssen nicht extra deklariert werden. Eine Custom Standard Action benötigt eine taglib Direktive. Die Taglib Direktive ordnet dem Prefix eine TagLibDescription Datei zu. Innerhalb einer TLD-Datei wird dem Aktionsname eine Klasse zugeordnet. Diese Klasse ist für die Ausführung der Aktion notwendig.

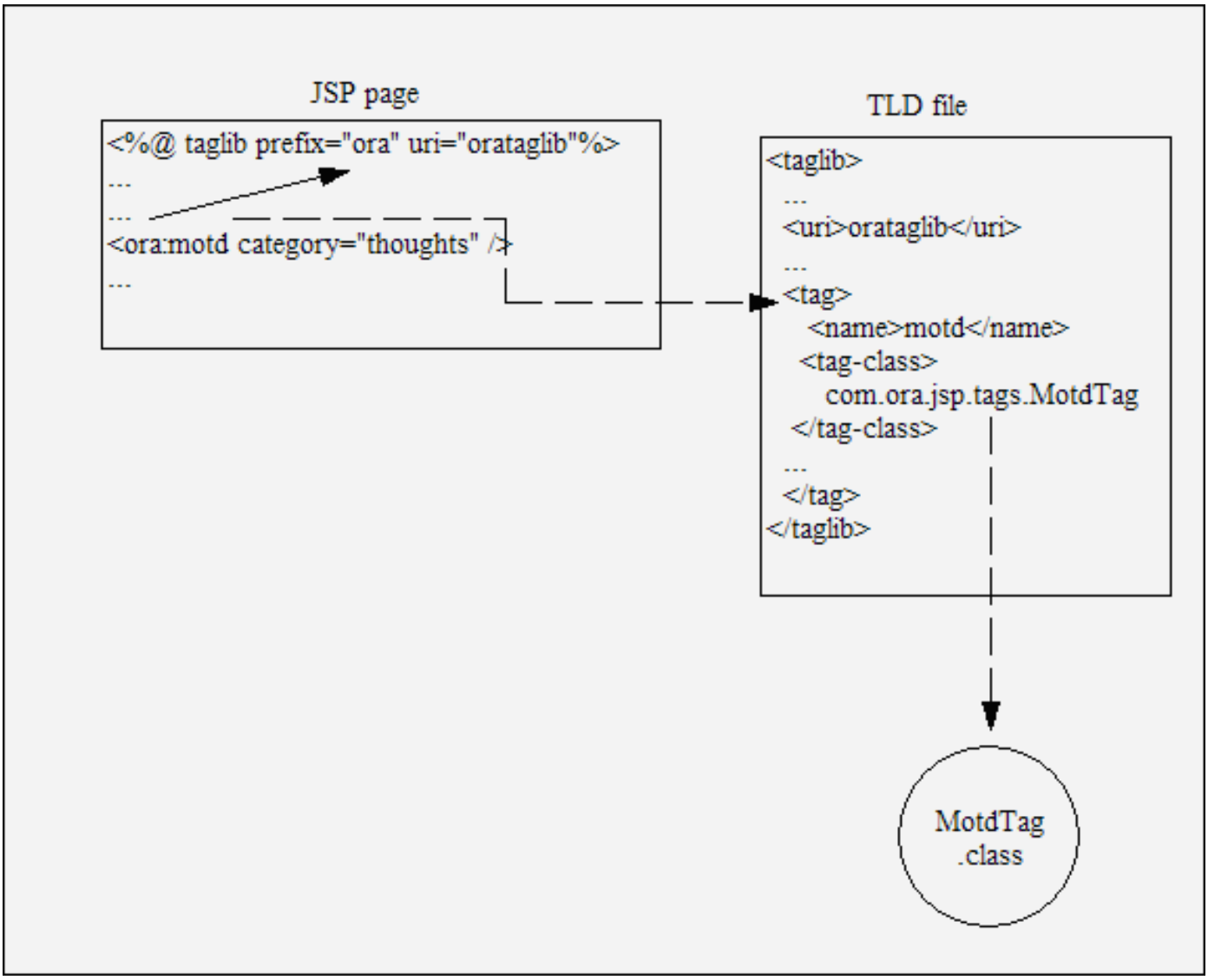


Abb. 8.1: Bedeutung der taglib Direktive

8.1.4 Expression Language Elemente

Die Expression Language wurde mit der neuen JSP 2.0 Definition eingeführt. Mit der EL ist es einfacher die dynamische Teile in die JSP einzubauen. Mit Hilfe von EL Ausdrücken können insbesondere Daten einfacher abgefragt werden, die in JavaBeans gespeichert sind. Gibt es in einer JavaBean beispielsweise ein Objekt *teilnehmer* mit einer Eigenschaft *nachname*, kann ein Wert einfach mit *${teilnehmer.nachname}* verwendet werden.

Man kann auch sehr einfach auf Elemente innerhalb einer Collection zugreifen z.B *${kurs.teilnehmer[0].nachname}*

Außerdem sin diese Ausdrücke für die Bestimmung von Attributwerten in Aktionen einsetzbar, wenn diese erst zur Laufzeit ermittelt werden sollen.

8.1.5 Zusammenfassung JSP Elemente

JSP Tag Type	Kurze Beschreibung	Tag Syntax
Directive	Spezifiziert zur Übersetzugszeit Anweisungen an die JSP Maschine	<%@ Directives %>
Deklaration	Deklariert und Definiiert Methoden und Variablen	<%! Java Deklerations %>
Scriplet	Ermöglicht es Java Code in JSP einzufügen	<% Some Java Code %>
Expression	Wird ausgeführt und die Auswertung in die Ausgabe eingefügt	<%= Code %>
Standard Actions	Enthält Laufzeit Anweisungen an JSP Maschine	<jsp:actionName />
Custom Actions	Laufzeit Anweisungen an JSP Maschine aus einzubindender Bibliothek	<namensraum:actionName />
Expression Language Elemente	Spezielle Sprache. Auswertung wird in Ausgabe eingefügt	${ ein EL-Ausdruck }
Comment	Wird nicht ausgewertet	<%-- Any Text --%>

8.2 Direktiven

Write JSP code that uses the directives: (a) 'page' (with attributes 'import', 'session', 'contentType', and 'isELIgnored'), (b) 'include', and (c) 'taglib'.

8.2.1 Page Direktive

Eine page Direktive informiert die JSP Maschine über Eingenschaften einer JSP Seite. Diese Eingenschaft betrifft nicht nur die aktuelle Seite, sondern auch die Seiten die mit der include Direktive eingebunden werden.

In der JSP 2.0 Spezifikation gibt es insgesamt 13 verschiedene Attribute, es werden lediglich nach der Objektive die folgenden Attribute für die Zertifizierung benötigt:

Attributname	**Beschreibung**	**Default Werte**
import	Eine Liste von Java Klassen und Packages, die in der JSP Seite gebraucht werden	Keine Default Werte
session	Falls *true* sollte die Seite Teil einer Session sein. Bei *false* ist zu dieser Seite keine implizite *session* Variable verfügbar	true
contentType	Enthält den Mime Type und character encoding der Ausgabe	text/html bzw text/xml
isELIgnored	falls true wird ein ${code} als normaler Text interpretiert und nicht ausgewertet	entsprechend Datei web.xml sonst false

Beispiele:

```
<%@ page import="java.util.*,java.text.*"%>
<%@ page contentType="text/plain"%>
```

Das import Attribut ist das einzigste Attribut innerhalb einer page Direktive die in einer Translation Unit mehrfach vorkommen können. Eine Translation Unit bezieht sich auf die akuelle JSP Seite und dijenigen die mit der include Direktive eingebunden werden.

8.2.2 Inlcude Direktive

Mit der Include Direktive wird zur Übersetzungszeit der JSP Seite in ein Servlet eine Datei eingebunden. Die include Direktive hat genau ein Attribut, nämlich *file* und enthält den Pfad zu einer Datei. Die Dateien die mit Include eingebunden werden gehören zu einer Translation Unit, d.h die aktuelle und eingebundene Dateien werden in ein Servlet übersetzt.

```
<html>
<head>
        <title>Including Files at Translation Time (JSP)</title>
</head>
<body>
        <%@ include file="somePage.jsp" %>
</body>
</html>
```

8.2.3 TagLib Direktive

Wird benötigt um custom Tags zu definieren

```
<%@ taglib prefix="example" uri="http://www.server.com/example-taglib" %>
```

In der Datei web.xml kann man den Pfad zur taglib Beschreibungs datei spezifizieren.

```
<taglib>
  <taglib-uri>http://www.server.com/example-taglib</taglib-uri>
  <taglib-location>/WEB-INF/example-taglib.tld</taglib-location>
</taglib>
```

Die Taglib-URI muss ein eindeutiger Name für die Tag Lib Bibliothek sein. Man kann auch den Umweg über die web.xml Datei umgehen und direkt den Pfad zur tld.Datei in der taglib Direktive angeben. Allerdings verliert man dadurch die Flexibilität die tld-Datei zu verschieben oder umzubenennen.

```
<%@ taglib prefix="example" uri="/WEB-INF/example-taglib.tld" %>
```

8.3 JSP als XML Dokumente

Write a JSP Document (XML-based document) that uses the correct syntax.

Es besteht die Möglichkeit JSP Seiten teilweise oder vollständig als XML-Dokumente anzulegen. Eine JSP Seite die vollständig als XML-Dokument angelegt ist (und damit wohlgeformt ist), nennt man JSP Dokument

8.3.1 Identifikation von JSP-Dokumenten

Der Servlet Container erkennt i.a. nicht automatisch ob es sich um eine normale JSP Seite oder um ein JSP-Dokument handelt. Folgende Verfahren zur Indentifikation von JSP-Dokumenten gibt es:

- Anstelle der sonst üblichen Dateierweiterung .jsp wird .jspx verwendent.
- Es wird zwar die Dateierweiterung .jsp verwendet, aber das oberste Element ist ein <jsp:root> Element. Ab der Spezifikation 2.0 muss das oberste Element eines JSP Dokumentes nicht *root* sein.
- Eine jsp Seite wird durch dir Datei web.xml als JSP-Dokument gekennzeichnet

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <is-xml>true</is-xml>
</jsp-property-group>
```

8.3.2 jsp:root Element

Ein JSP-Dokument beginnt i.a. nicht mit dem typischen xml Kopf *<?xml version="1.0" ?>* Wird ein jsp:root Element verwendet, so muss das oberste Element xml sein. Es hat zwingend das Attribut version, dessen Werte "2.0" oder "1.2" sein musss. Ausserdem wird es benutzt um Namensräume zu deklarieren.

```
<jsp:root
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   version="2.0">
```

```
   ...
</jsp:root>
```

Die Namespaces werden verwendet um Tag-Bibliotheken der Aktionen zu deklarieren.

8.3.3 Kommentare

Es gibt keine spezielle Tags für Kommentare um einen Kommentar von einer JSP-Seite umzuwandeln sollte man einen normalen xml Kommentar verwenden.

```
<%-- Dies ist ein JSP Kommentar --%>
<!-- Dies ist ein xml Kommentar -->
```

8.3.4 Template Text

Template Texte werden durch das Element *jsp:text* deklariert. Man muss aufpassen dass durch den Template Text die XML Syntax nicht verletzt wird.

Beispiel eines ungültigen Template Textes

```
<jsp:text>
  bestanden? $(test.punktzahl > 1000)
</jsp:text>
```

Das > Zeichen ist innerhalb eines xml-Elements nicht erlaubt. Ersetze in diesem Fall das > Zeichen durch gt

8.3.5 Skriptelemente

Deklarationen

Es wird das Element *jsp:declaration* verwendet. Dieses Element hat keine Attribute

```
<%!
        public String someFunc(int i) {
                if (i<3) return("...");
        }
%>

<jsp:declaration>
        public String someFunc(int i) {
                if (i&lt;3) return("...");
        }
</jsp:declaration>
```

Skripte

Es wird das Element *jsp:scriptlet* verwendet. Dieses Element hat keine Attribute

```
<% out.println("Gesamtbetrag: "); %>

<jsp:scriplet>
    out.println("Gesamtbetrag: ");
</jsp:scriplet>
```

Expressions

Es wird das Element *jsp:expression* verwendet. Dieses Element hat keine Attribute

```
2 + 3 = <%= 2 + 3 %>

<jsp:text>
    2 + 3
```

```
</jsp:text>

<jsp:expression>
   2 + 3
</jsp:expression>
```

8.3.6 Direktiven

Bei Page Direktiven wird das Element *jsp:directive.page* verwendet. Bei Include Direktiven wird das Element *jsp:directive.include* verwendet.

Beispiel:

```
<jsp:directive.page contentType="text/html" />
<jsp:directive.include file="anotherFile.jsp" />

Es gibt kein xml Elmement für die taglib Direktiven. Die Spezifikation der taglib
Bibliotheken, erfolgt stattdessen durch Namensräume.

$S()
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:forEach items="$(paramValus}" var="parValue">
  ....
  </c:forEach>
```

Eine Namensraumdeklaration kann auch mitten in einem Element stehen, und gilt dann für dieses Element und den untergeordneten Elementen.

```
<c:forEach xmlns:c="http://java.sun.com/jsp/jstl/core"
    items="$(paramValus}" var="parValue">
  ....
  </c:forEach>
```

8.4 Der JSP Life Cycle

Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the jspInit method, (6) call the _jspService method, and (7) call the jspDestroy method.

8.4.1 Die 7 Phasen eines JSP Lebenszyklus

Eine JSP Seite durchläuft folgende 7 Phasen

Phase	**Beschreibung**
Übersetzung	Die JSP Seite wird geparst und es wird daraus eine Java-Datei mit dem entsprechenden Servlet erzeugt
Kompilieren	Die erzeugte Datei wird compiliert
Klasse einladen	Die compilierte Klasse wird eingeladen
Instantiierung	Es wird eine Instanz eines Servlets erzeugt
jspInit()	Diese Methode ermoeglicht die Initialisierung des Objekts
_jspService()	Diese Methode wird für jede Anfrage ausgeführt

jspDestroy()	Diese Methode wird aufgerufen, bevor der Container die Instanz dieses Servlet zerstört

Die Übersetzung

Zunächst wird überprüft, ob sich die JSP seit der letzten Übersetzung geändert hat. Falls ja finden einige Überprüfungen statt, und es werden die Direktiven ausgewertet. Das Ergebnis ist eine Datei mit Java Code. Der Name der Klasse ist abhängig vom verwendeten Container.

Kompilieren

Die erzeugte Java Klasse wird mit einem normalen Java Compiler compiliert.

Laden und Instantiieren

Die geladene Instanz sollte (falls Http verwendet wird) das folgende Interface HttpJspPage erfüllen:

```
public interface JspPage extends Servlet {

        public void jspInit();
        public void jspDestroy();
        public void _jspService(ServletRequestSubtype request,
                ServletResponseSubtype response)
                throws ServletException, IOException;
                // _jspService - depends on the specific protocol used and
                // cannot be expressed in a generic way in Java.
}

public interface HttpJspPage extends JspPage {

   public void _jspService(HttpServletRequest req, HttpServletResponse res)
                throws javax.servlet.ServletException, java.io.IOException;

}
```

Die Methode jspInit()

Diese Methode wird direkt nach der Instaniierung, auf jedenfall vor der ersten Anfrage ausgeführt. Man kann diese Methode als Skriplet in die JSP Seite deklarieren.

```
<%!
        public void jspInit() {
                ...
        }
%>
```

Beachte man ist nicht gezwungen, diese Methode zu deklarieren. Sie existiert schon in der Basisklasse.

Die Methode jspDestroy()

Diese Methode wird aufgerufen, wenn der Container entschlossen hat, das entsprechende Servlet auszuladen. Auch diese Methode muss nur dann deklariert werden, wenn man sie wirklich benötigt.

```
<%!
        public void jspDestroy() {
                ...
        }
%>
```

Die Methode _jspService()

Diese Methode wird automatisch bei jeder Anfrage aufgerufen. Diese Methode wird bei der Übersetzung der JSP in das Servlet automatisch generiert. Es führt zu einem Fehler wenn der Programmierer versucht diese Methode zu deklarieren.

8.4.2 Translatation Unit

Mit Hilfe der include Direktive kann man den Inhalt einer anderen Seite (statisch oder wieder eine JSP) einbinden. Es wird nicht nur die aktuelle Seite, sondern auch die eingebundenen Seiten in ein einziges Servlet übersetzt. Die Menge an Seiten die in ein einziges Servlet übersetzt werden, wird Translation Unit genannt.

Für eine Translation Unit gilt einige Regeln:

- Die Page Direktiven beziehen sich auf die ganze Translation Unit.
- Eine Variable Deklaration oder eine Methoden Dekleration darf nicht innerhalb einer Translation Unit mehrmals vorkommen.
- Die Standard Aktion *jsp:useBean* kann nicht innerhalb einer Translation Unit zweimal dieselbe JavaBean deklarieren.

8.4.3 Übersetzung einer JSP im Detail

- Die Direktiven werden während der Übersetzung ausgewertet. Einige Direktive wie die page Direktive mit Attribut import erzeugen Code in der ServletKlasse. Andere Direktiven setzen Eigenschaften einer Seite fest, und führen nicht zu einem Code in das Servlet.
- Alle Deklarationen werden in die zu übersetzende Servlet Klasse eingebaut, Variablen Deklarationen werden zu Instanzvariablen umgewandelt, Methoden Deklarationen werden Methoden des Servlets.
- Alle JSP Scriplets werden Teil der _jspService() Methode.Variablen innerhalb eines Skriplets werden zu lokalen Variablen der _jspService() Methode.
- Auch alle Ausdrücke und EL Ausdrücke werden Teil der _jspService() Methode. Für EL Ausdrücke werden durch spezielle Klassen ausgewertet. Die herkömmlichen JSP Ausdrücke werden durch eine out.print() Methode umwickelt.
- Alle JSP Aktionen werden durch Aufrufe an entsprechende Klassen ersetzt
- Ausdrücke (EL oder ordinär) die innerhalb einerhalb einer Aktion einem Attribut einen Wert zuweisen, werden nicht mit out.print ausgegeben. Der Ausdruck wird in diesem Fall während einer Anfrage ausgewertet, dem Attribut zugewiesen und der Aktion entspechenden Java Klasse aufgerufen.
- Alle JSP Kommentare werden ignoriert.
- Die Template Texte werden Teil der _jspService() Methoden und in die Methode out.write() umwickelt.

Beachte einige Besonderheiten:

- Es ist nicht wichtig in welcher Reihenfolge die Deklarationen in der JSP Seite erscheinen.

Beispiel:

```
<html>
<body>
   pi = <%=pi%>
<%! final double pi=3.1414 %>
</body>
</html>
```

Das Beispiel ergibt kein Fehler obwohl die Deklaration von pi erst nach der Benutzung erfolgt.

- Bei Skriplets spielt die Reihenfolge der Variablendeklarationen sehr wohl eine Rolle

```
<html>
<body>
<%  String s = s1+s2; %>
<%! String s1 = "hello"; %>
<%  String s2 = "world"; %>
<% out.println(s); %>
</body>
</html>
```

Dieses Beispiel scheitert in der Zeil *String s = s1+s2;*. Das Problem liegt nicht in der Varialbe s1, da diese in einer Deklaration ist, und in eine Instanzvariable übersetzt wird. Das Problem liegt in der lokalen Variablen s2, die zu spät deklariert und definiert wird.

- Iterative und Conditionale Ausdrücke

Man sollte darauf achten dass bei Schleifen, immer in Skriplets immer die Klammern gesetzt werden, auch dann wenn man scheinbar nur eine Anweisung hat.

Siehe z.B folgenden JSP-Code:

```
<% if (isUserLoggedIn) %>
Welcome, <%= userName %>
```

Dies wird folgendermassen übersetzt:

```
if (isUserLoggedIn)
        out.write("Welcome, ");
    out.print(username);
```

Der Anwendername wird auch angegeben, wenn isUserLoggedIn den Wert false hat.

So wäre es richtig:

```
<% if (isUserLoggedIn)
   {
%>
  Welcome, <%= userName %>
<%
   }
%>
```

8.5 Implizite JSP Variablen

Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.

8.5.1 Übersicht

Die JSP Maschine legt in der Methode _jspService() automatisch einige Variablen an. Hier eine Übersicht der möglichen impliziten Variablen

Identifier name	**Klasse oder Interface**	**Beschreibung**
application	interface javax.servlet.ServletContext	Zeigt auf die die Umgebung der

		Applikation
session	interface javay.servlet.http.HttpSession	Beinhaltet die Session eines Benutzers
request	interface javax.servlet.http.HttpServletRequest	Enthält die Anfrage auf die Seite
response	interface javax.servlet.http.HttpServletResponse	Zum Senden der Antwort an den Client
out	class javax.servlet.jsp.JspWriter	Enthält den Output Stream der aktuellen Seite
page	class java.lang.Object	Ist die Servlet Instanz der aktuellen Seite
pageContext	class javax.servlet.jsp.PageContext	Umgebung der aktuellen Seite
config	interface javax.servlet.ServletConfig	Konfiguration eines Servlet
exception	class java.lang.Throwable	Benutzt für Fehlerbehandlung

Das folgende Listing zeigt wie diese impliziten Variablen in die Methode _jspService() eingebaut wird

```
public void _jspService(
          HttpServletRequest request,
          HttpServletResponse response
              throws java.io.IOException, ServletException
{
    ... other code
PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
... other code
pageContext = .... // getIt from Somewhere
    session = pageContext.getSession();
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    out = pageContext.getOutput()
... other code
}
```

8.5.2 Implizite Variablen

request, response

Dies sind HttpServletRequest bzw. HttpServletResponse Objekte die bei einer Anfrage, an die _jspService Methode übergeben werden. Sie haben genau dieselbe Funktionalität wie bei den Servlets.

out

Diese ist vom Typ javax.servlet.jsp.JspWriter. Man benutzt die print() bzw. println() Methode von out, um an die Antwort die zum Clienten gesendet wird Objekte bzw. Text

anzuhaengen.JspWriter ist eine Unterklasse vom java.io.Writer Objekt und versteht all die print() bzw println() Methoden um z.B boolesche oder numerische Werte auszugeben.

session

Diese Variable existiert nicht falls mit einer page-direktive das Attribut session auf false gesetzt wird (<%@ page session="true" %>)

Ansonsten besteht die implizite Variable session aus dem HTTP Session Objekt, das man auch durch den Befehl request.getSession() bekommt.

config

Diese Variable besteht aus dem ServletConfig() Objekt, mit dessen Hilfe man die Initialien Parameter aus der Deployment Descriptor Datei auslesen kann.

application

application entspricht dem javax.servlet.ServletContext Objekt der WebApplikation.

page

Page wird sehr selten verwendet, und entspricht der aktuellen Servlet Instanz. Es ist identisch mit

```
Object page = this   //this refers to the instance of this Servlet
$\

Beachte das page vom Typ Object ist, und deshalb nicht direkt die Servlet Methoden kennt.

$S()
<%= page.getServletInfo() %>  Ergibt einen Fehler
<%= ((Servlet) page).getServletInfo() %>  Mit Casten ist es Ok
<%= this.getServletInfo() %>   Auch ok
```

pageContext

pageContext enthält ein konkrete Unterklasse von javax.servlet.jsp.PageContext(). Diese hat folgende Aufgaben

- Sie ist ein Container für viele der implizite Variablen

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

- Enthält Methoden um Attribute in verschieden Scopes zu speichern. Die Scopes einer JSP Seite werden weiter unten erläutert
- Beinhaltet Methoden um Anfragen an eine andere Ressource weiterzuleiten

Methode	**Beschreibung**
void include(String relativeURL)	Die Ausgabe einer anderen Ressource wird in die aktuelle Seite eingebunden. Dies ist identisch mit aServletRequest.getRequestDispatcher().include()
void forward(String relativeURL)	Leitet die Anfrage an eine andere Ressource weiter. Dies ist identisch mit aServletRequest.getRequestDispatcher().forward()

exception

Diese implizite Variable ist nur sichtbar falls die aktuelle Seite als errorPage deklariert ist. exception ist vom Typ java.lang.Throwable und enthält Informationen über die geworfene Exception.

8.5.3 Die Page Scopes

Die folgende Abbildung verdeutlicht die Scopes einer JSP Seite

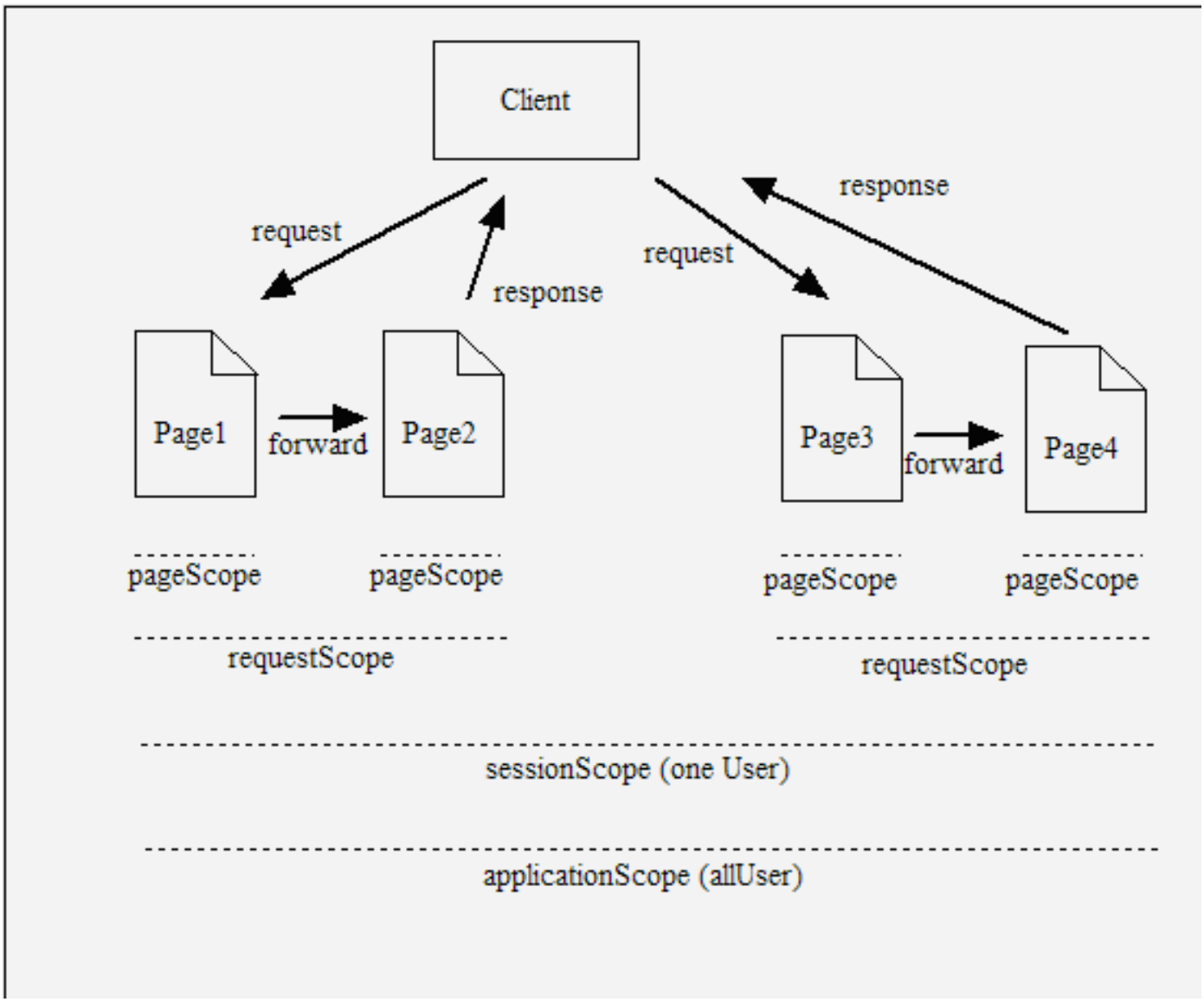


Abb. 8.2: Scopes einer JSP

Es existieren die 4 folgenden Scope

Scope Name	Gültigkeit
application	Gültig innerhalb einer Web-Application
session	Limitiert zu einer Session eines einzelnen Benutzers
request	Limitiert zu einer einzelnen Anfrage ggf. ueber mehrere Seiten

page	Limitiert zu einer einzelnen Seite und einer einzelnen Anfrage

Mit Hilfe dem PageContext Objekt lassen sich die Attribute für einen Scope setzen bzw. auslesen. Dazu sind einige Integer-Konstanten für die einzelnen Scopes definiert

- static final int APPLICATION_SCOPE
- static final int SESSION_SCOPE
- static final int REQUEST_SCOPE
- static final int PAGE_SCCOPE

Folgende Methoden dienen zum Setzen von Attributen in einem Scope

- void setAttribtue(String name, Object object, int scope);
- Object getAttribute(String name, int scope);
- void removeAttribute(String name, int scope);
- Enumeration getAttributeNamesInScope(int scope);

Es gibt auch Methoden zum Auffinden Attribute wenn man nicht weiss in welchen Scope sie definiert sind

Object findAttribut(String name)	sucht ein Attriubut zuerst in page, dann request, dann session (falls gültig) dann application Scope.
int getAttributesScope(String name)	Gibt den Scope zurück in der ein Attribut definiert ist

8.6 Einbinden von tld.Dateien und Deaktivierung EL bzw. Scripting

Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language.

8.6.1 Zuordnung URI zur tld Datei

Mit Hilfe von Tag Bibliotheken kann man die Sprache mit der man JSP Seiten gestalten kann nach seinen eigenen Bedürfnissen erweitern. Man kann eigene Aktionen definieren, oder Funktionen für die Expression Language schreiben.

Die tld Datei beschreibt die Schnittstelle zu den vom Programmierer definierten Funktionen bzw. Aktionen. Hier geht es zur Verbindung der Deployment Descriptor Datei web.xml zu einer tld Datei.

Jede tld Datei hat eine URI, die sie eindeutig identifiziert. Zwei tld-Dateien können nicht dieselbe URI haben.

Bei einer vom Programmierer definierten Funktion, muss die JSP-Maschinen die tld die zugehörende tld Datei finden, und anhand der dort beschriebenen Schnittstelle die entsprechenden Java Klassen aufrufen.

Die Zuordnung einer benutzerdefinierten Aktion bzw. Funktion zu einer tld Datei geht über das Prefix und die taglib Direktive.

```
<%@ taglib prefix="myPrefix"  uri="http://www.myCompany.aTagLibrary" %>

...

<myPrefix:eineAktion attribut1="blabla" >
...
</myPrefix:eineAktion>
```

Anhand der URI alleinig erkennt der ServletContainer nicht unbedingt, wo die entsprechende tld-Datei abgelegt ist.

Es wird zwischen expliziten und impliziten Zuordnung der tld-Datei zu einer URI unterschieden.

implizites Mapping

Alle tld Dateien, die sich bei den gepackten *.jar befinden, werden automatisch gelesen. Enthalten diese tld Dateien eine URI erfolgt die Zuordnung einer URI zu der Datei automatisch.

explizites Mapping

Es gibt 3 verschiedene Arten von URIs in der taglib Direktive einer JSP Seite

Typ	Beschreibung	Beispiele
Absolute URI	Die URI hat ein Protokoll, Hostname und ggf. eine Portnr	http://localhost:8080/taglibs http://www.manning.com/taglibs
Root relative URI	Die URI startet mit einem Slash / hat kein Protokoll und Hostname	/helloLib /taglib/hellolib
Nicht Root relative URI	Die URI beginnt nicht mit einem Slash und hat kein Protokoll Hostname oder Portnummer	HelloLib taglib/helloLib

- absolute URI

Bei einer absoluten URI wird der Ort der TLD Datei in der Deployment Descriptor Datei gesucht

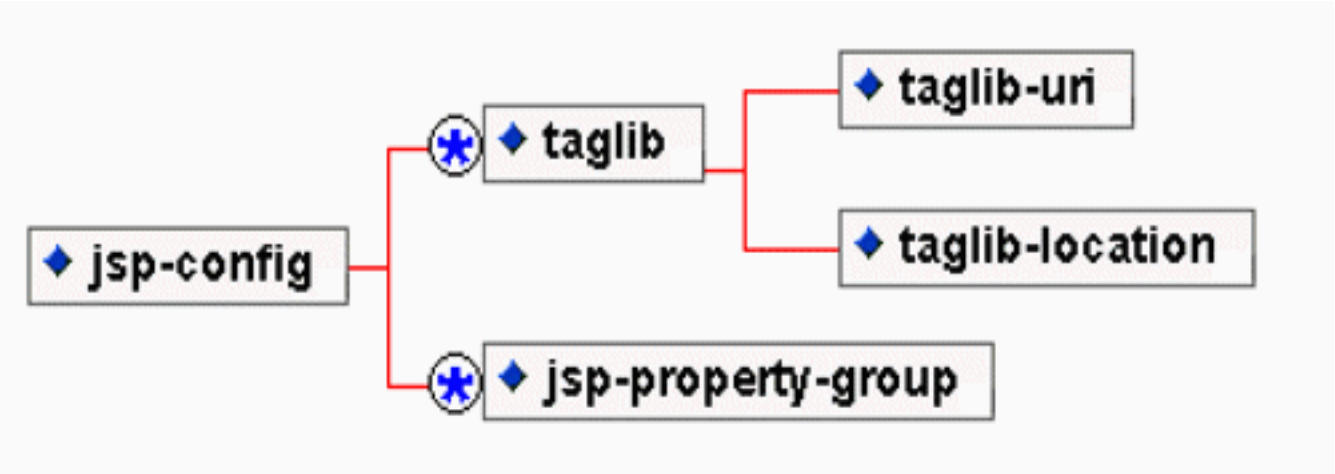


Abb. 8.3: tagLib Zuordnung

Der Zuordnung des Pfads der tld-Datei zu einer URI steht innerhalb des taglibs Elements, das in einem jsp-config Element eingebunden ist. Beginnt der Pfad mit einem Slash / so wird dieser Pfad relativ zum obersten Ordner der root Applikation gesucht, ansonsten wird der Pfad relativ zur aktuellen jsp-Seite gesucht.

- Root Relative URI

Die TLD steht an dem in der URI angegebenen Pfad relativ zum obersten Ordner der Web Applikation

- Nicht Root Relative URI

Die TLD steht an dem in der URI angegebenen Pfad relativ zur aktuellen JSP Seite

8.6.2 Aktivierung bzw. Deaktivierung EL-Ausdrücke bzw Skripting

In der Datei web.xml kann man die Expression Language für alle oder einzelne JSP-Seiten abschalten.

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
</jsp-property-group>
$S

Mit Einführung der EL sind Skripting Elemente innerhalb jsp-Seiten unnötig, und man kann
in der web.xml konfigurieren, dass diese für bestimmte oder alle JSP Seiten abgeschaltet
werde.

$S()
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

Das Element <jsp-property-group> ist ein Unterelement von <jsp-config>.

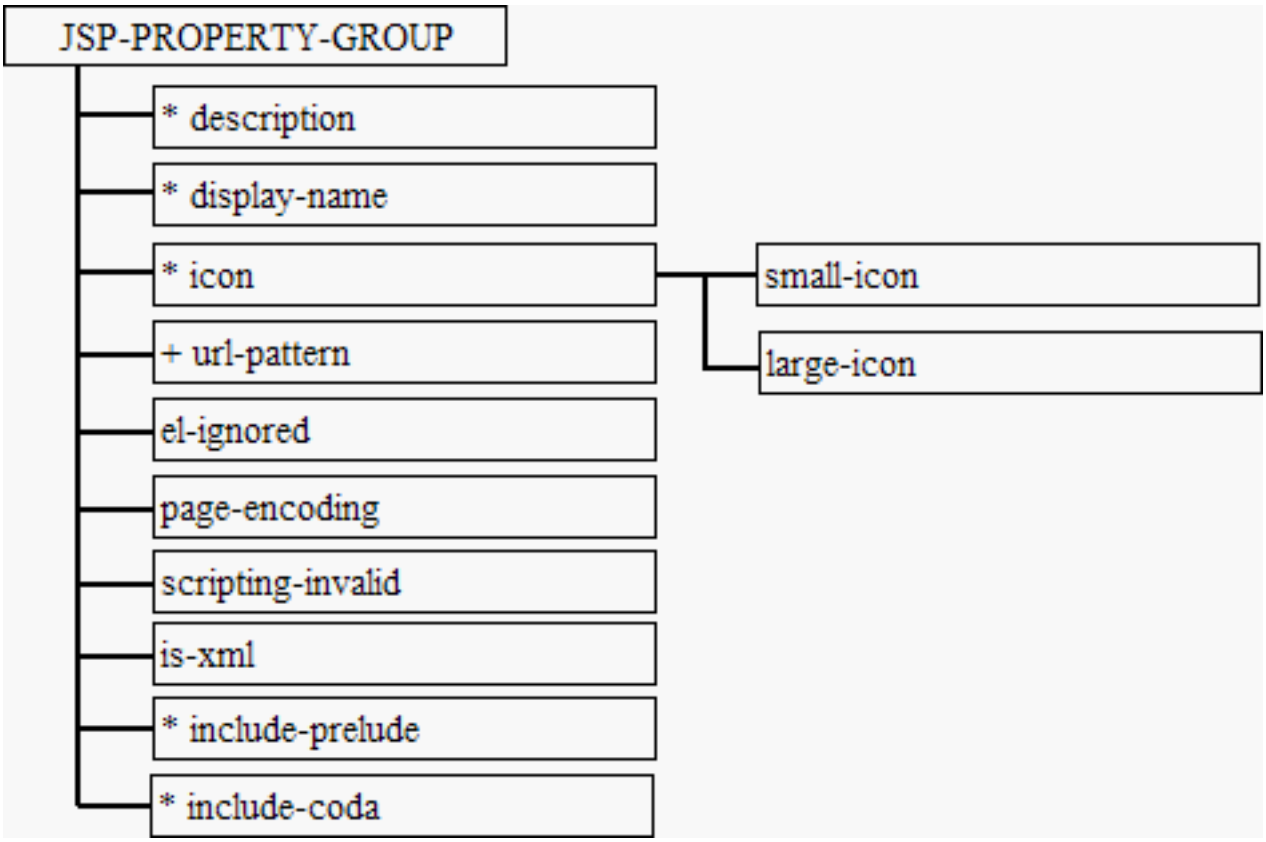


Abb. 8.4: Eigenschaften einer JSP definieren

Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the include directive or the jsp:include standard action).

8.7 Inkludieren von Dateien

Die Include Directive

Mit der Inlcude Direktive wird zur Übersetzungszeit (von JSP in ein Servlet) eine Datei in die JSP Seite eingebunden.

Beispiel:

```
<html>
        <head><title>An Include Test</title></head>
        <body bgcolor="white">

                The current date and time are :

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

        </body>
</html>


date.jsp:

<%= (new java.util.Date() ).toLocaleString() %>
```

Die <jsp:include> Aktion

Es wird nicht die Zieldatei selbst, sondern die Ausgabe der Zieldatei in die Ausgabe der aktuellen Datei eingebunden. Im Gegensatz zur include Direktiven erfolgt die Einbindung zur Laufzeit.

Es ist mit dem <jsp.param> Element möglich Parameter zu übergeben, die Parameterwerte

müssen nicht konstant sein.

Beispiele:

```
<jsp:include page = "eineAndereSeite.jsp">

     <jsp:param name="username" value="Hansen" />
 </jsp:include>
```

```
<jsp:include page = <%= request.getParameter("incFile")%> flush="true" >
      <jsp:param name="username" value= ${request.username} />
  </jsp:include>
```

Ausserdem kann man der include Seite optional einen booleschen Parameter flush mitgeben. Der DefaultWert ist false. Bei true wird vor der Übernahme der aufgelaufen Inhalt des Puffers an den Client geschickt und der Puffer geleert.

Implizite Inlcudes Definitionen

Es ist möglich deklarativ automatisch Seiten mit einer include Direktive einzubinden. Mit include-prelude werden automatisch Dateien am Beginn einer JSP Seite eingebunden, und mit include-code wird automatisch eine Datei am Ende einer JSP Seite eingebunden.

Die Endung jspf kennzeichnet im unteren Beispiel, dass es sich um ein Fragment einer JSP Seite handelt. Allerdings sollte auch innerhalb eines Fragments zu jedem StartTag ein EndTag existieren.

```
<jsp-config>
  <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <include-prelude>/WEB-INF/jspf/prelude1.jspf</include-prelude>
        <include-coda>/WEB-INF/jspf/coda1.jspf</include-coda>
  </jsp-property-group>
  <jsp-property-group>
        <url-pattern>/two/*</url-pattern>
        <include-prelude>/WEB-INF/jspf/prelude2.jspf</include-prelude>
        <include-coda>/WEB-INF/jspf/coda2.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

9 Die Expression Language

9.1 Zugriff auf implizite Variablen

Given a scenario, write EL code that accesses the following implicit variables including: pageScope, requestScope, sessionScope, and applicationScope, param and paramValues, header and headerValues, cookie, initParam and pageContext.

Mit der Expression Language hat man Zugriff auf eine Reihe von implizierten Objekten, die in der folgenden Tabelle zusammengefasst sind.

Name	Bedeutung
pageContext	Die Instanz der PageContext Klasse ,für die aktuelle Seite
pageScope	Eine Collection (java.util.Map) aller page scope Variablen
requestScope	Eine Collection (java.util.Map) aller request scope Variablen
sessionScope	Eine Collection (java.util.Map) aller session scope Variablen
applicationScope	Eine Collection (java.util.Map) aller application scope Variablen
param	Eine Collection (java.util.Map) aller Parameterwerte einer Anfrage.${param.name} entspricht aServletRequest.getParameter(name)
paramValues	Eine Collection (java.util.Map) aller Parameterwerte einer Anfrage als String Array. ${paramValue.name} entspricht aServletRequest.getParameterValues(name)
header	Eine Collection (java.utilMap) aller Header Werte einer Anfrage.
headerValues	Eine Collection (java.utilMap) aller Header Werte einer Anfrage als String Array
cookie	Eine Collection (java.utilMap) von Cookie-Namen zu einzelnen Cookie Werten
initParam	Eine Collection (java.utilMap) aller Initialisierungsparameter

Beispiele:

- **${param["userName"]}** Dies gibt der Parameterwert von userName einer Anfrage zurück. Die Anfrage ist identisch mit **${param.userName}**.
- **${pageContext.request.requestURI}** Die URI der Anfrage
- **${sessionScope.cart.numberOfItems}** Das Propertie numberOfItems aus dem Attribut das unter dem Namen cart in der sessionScope abgespeichert ist.
- **${header["host"]}** Wert des Paramters Host vom Header. Dies ist identisch mit **${header.host}**

Bei einem Ausdruck ${einString} wird erstmal geschaut ob *einString* ein implizites Objekt ist. In diesem Falle wird dieses zurückgegeben. Ansonsten wird mit der Funktion *PageContext.findAttribute(einString)* der Paramter *einString* in den Scopes gesucht. Es werden nacheinander die Scopes *page*,*request*,*session* und *application* durchsucht bis das Attribut *einString* gefunden wird. Es wird der Wert des Attributs *einString* zurückgegeben, oder falls es diesen nicht gibt, wird der Wert *null* zurückgegeben. Sollte es ein Attribut innerhalb eines Scopes wie ein implizites Objekt heissen, so wird das implizite Objekt zurückgegeben.

9.2 Zugriff auf Bean Properties und Collection Elemente

Given a scenario, write EL code that uses the following operators: property access (the '.' operator), collection access (the '[]' operator).

Mit dem Punkt Operator '.' wird auf die Properties eines Beans zugreifen.

```
${aBean.aProperty.aPropertyOfTheProperty.andSoOn}
```

Auch ist es sehr einfach, auf eine Map zuzugreifen:

```
${aMap.aKey}
```

Ausdrücke mit der Syntax *'${identifier[subexpression]}'* werden folgendermassen ausgewertet:

- Werte den identifier und den subexpression aus; wenn einer von beiden null ergibt ist der Gesamtausdruck null
- Falls es sich beim identifier um eine BEAN handelt: Die subexpression wird als String umgewandelt. Dieser String wird als eine Bean Property Eigenschaft angesehen. Als Ergebnis wird der Wert der Propertie ausgegeben. So wird für ${name.["lastName"]} der Wert der Methode *name.getLastName()* ausgewertet.
- Falls es sich beim Identifier um ein ARRAY handelt: Die subexpression wird zu einem int Wert umgewandelt. Das Ergebnis des Gesamtausdrucks ist dann der Wert des entsprechenden Elements des Arrays. z.B: colors["3"] ist identisch mit colors[3].
- Falls es sich beim Identifier um eine Liste handelt: Dies ist identisch mit dem Umgang zu einem Array. Die subexpression wird als Integer umgewandelt, und der Gesamtausdruck enthält das entsprechende Element der Liste.
- Falls es sich beim identifier eine MAP handelt. Die subexpression wird als key einer Map angesehen. Sie wird nicht umgewandelt, da als key jedes Objekt sein kann. Ist subexpression kein String, wird versucht den key in der in einer der Scopes der JSP zu finden.

Identifier type	**Example use**	**Method invoked**
JavaBean	${colorBean.red} ${colorBean["red"]} ${colorBean['red']}	colorBean.getRed()
Array	${colorArray[2]} ${colorArray["2"]}	Array.get(colorArray, 2)
List	${colorList[2]} ${colorList["2"]}	colorList.get(2)
Map	${colorMap[red]}	colorMap.get(pageContext.findAttribute("red"))
	${colorMap["red"]} ${colorMap.red}	colorMap.get("red")

9.3 Operatoren

Given a scenario, write EL code that uses the following operators: aritmetic operators, relational operators, and logical operators.

9.3.1 Arithmetische Operatoren

Es existieren die gewohnten arithmetischen Operatoren: '+', '-', '*', '/', '%'. Für die Division kann man ausser '/' auch *div* benutzen. Für Modulo kann man ausser '%' auch *mod* benutzen.

```
6 + 7 = ${6+7}
8 x 9 = ${8*9}
```
Bei einer Division durch 0, wird keine Exception geworfen sondern der String 'Infinity' zurückgegeben.

9.3.2 Relationale Operatoren

Die relationalen Operatoren sind in folgender Tabelle zusammengefasst:

Symbol version	**Text Version**
==	eq
!=	ne
<	lt
>	gt
>=	ge
<=	e

Hier sind einige Beispiele:

```
Is 1 less than 2? ${1<2} <br>
Does 5 equal 5? ${5==5} <br>
Is 6 greater than 7? ${6 gt 7} <br>
```

9.3.2.1 Logische Operatoren

Die logischen Operatoren entsprechen denen von Java. Für jeden operator gibt es auch eine textuelle Variante:

Symbol	**version Text Version**
&&	and
ll	or
!	not

Da es in EL kein null Wert gibt, wurde der *empty* Operator eingefügt.

Beispiel

Dieser Operator wird true fall:

- Eine Objketreferenzen nicht existiert, d.h. null ist
- Ein String nicht existiert oder von der Länge 0 ist
- Ein Array nicht existiert oder von der Länge 0 ist
- Eine Liste nicht existiert oder von der Länge 0 ist
- Eine Map nicht existiert bzw. keine Einträge hat.

Der Operator wird folgendermassen aufgerufen:

```
empty variableName
```

9.4 Funktionen nutzen und definieren

Given a scenario, write EL code that uses a function; write code for an EL function; and configure the EL function in a tag library descriptor.

Die Expression Language lässt sich um eigene Funktionen erweitern. Eine Funktion besteht aus einem Präfix, dem Funktionsnamen und ggf. einigen Parametern.

```
ns:func(a1, a2, ..., an)
```

Es muss eine taglib Direktive existieren, die dem Prefix (ns) eine TagLibDescripiton Datei zuordnet.

z.B:

```
<%@ taglib prefix="ns" uri="http://myUrl.com/some-taglib" %>
```

In dieser TLD Datei wird die Schnittstelle der Funktionen definiert.

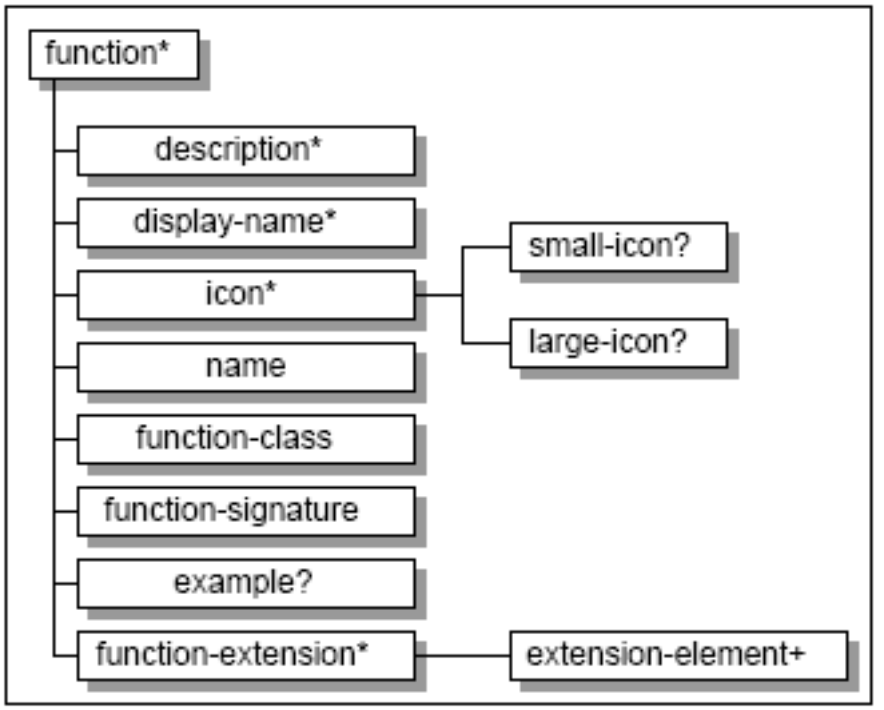


Abb. 9.1: Definieren einer eigenen EL Funktion

Die Java Klasse muss in der tld-Datei voll qualifiziert angegeben werden. Die eigentliche Funktion ist eine statische Funktion der angegebenen Java Klasse.

Die Bedeutung von *function-extension* und *extension-element* ist mir selbst noch unklar, wird aber wahrscheinlich für die Zertifizierung nicht benötigt.

Hier ein Beispiel einer selbstgeschriebenen concat-Funktion

```
<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>
        <function>
                <description>Concatenates two strings</description>
                <name>concat</name>
                <function-class>com.example.MyELFunctions</function-class>
                <function-signature>
                        java.lang.String concat(java.lang.String, java.lang.String)
                </function-signature>
        </function>
</taglib>
```

```
package com.example;

public class MyELFunctions {
        public static String concat(String str1, String str2) {
                return str1 + str2;
        }
}
```

Tritt beim Ausführen der Funktion eine Exception auf, so wird diese Exception in eine ELException gewrapped, und diese ELException wird geworfen.

10 Standard Aktionen

10.1 Die Aktion <jsp:useBean>

Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: 'id', 'scope', 'type', and 'class'), jsp:getProperty, and jsp:setProperty (with all attribute combinations).

Eine Java Bean ist eine normale Java Klasse. Es gibt die Einschränkung, dass eine JavaBean ein parameterloser Konstruktor hat. Dies ermöglicht ein beliebiges Tool alleinig mit Wissen des Klassennamens die Java Bean zu instantiieren. Eine Java Bean hat Properties. Diese werden mit getter und setter Methoden gelesen bzw. definiert.

Das folgende Beispiel zeigt die Benutzung einer JavaBean.

```
<jsp:useBean id="connection" class="com.myco.myapp.Connection" scope="page"/>
```

Zunächst wird gesucht, ob es im Scope page ein Attribut mit dem Namen *connection* gibt. Falls es das Attribut nicht gibt, wird die Klasse *com.myco.myapp.Connection* instantiiert und mit dem Namen *connection* dem entsprechenden scope zugeordnet. Danach kann man mit dem Expression Language ${connection} auf die Instanz zugreifen.

Statt der Klasse kann man auch das *beanName* angeben. Allerdings ist es nicht möglich gleichzeitig das Attribut *beanName* und *class* anzugeben, aber mindestens eines der beiden Attribute muss angegeben sein. Bei Angabe des Attributs *beanName* erfolgt die Instantiierung über die Klasse *java.bean.Beans* mit der Methode *instantiate()* und den Wert von *beanName* als Argument.

Die Aktion <jsp:useBean> kann auch ohne eines der Attribute *class* oder *beanName* angewendet werden.

```
<jsp:useBean id="connection" type="com.myco.myapp.Connection" scope="page"/>
```

Es wird im scope page ein Bean gesucht, der die Klasse oder Interface *"com.myco.myapp.Connection"* erfüllt und der id *connection* zugeordnet. Wird dieses Bean nicht gefunden, so wird keines erzeugt.

Attribut	**Beschreibung**
id	Name der die JavaBean im angegebenen Scope identifiziert und zugleich SkriptVariablenname für die Referenz auf das Objekt.
scope	Geltungsbereich, in dem die Objektreferenz verfügbar ist. Vorgabe ist *page*
class	Voll qualifizierter Klassenname
beanName	ame einer Bean, wie er von der Methode *instantiate()* in der Klasse *java.beans.Beans* erwartet wird.
type	bestimmt den Typ der Skriptvariablen unabhängig vom Typ der Implementierungsklasse. Als Typ kann die Klasse, eine Oberklasse der Klasse oder ein Interface, das die Klasse implementiert, angegeben werden. Vorgabe ist der Wert von Attribut *class*

10.2 Die Aktion <jsp:getProperty>

Die Aktion *<jsp:getProperty>* greift auf die Eigenschaften der vorher deklarierten JavaBean-Komponente zu. Das Ergebnis der Aktion wird, wenn nötig in einen String umgewandelt, in die Antwort eingefügt.Bei Objekten wird zur Umwandlung die *toString* Methode verwendet, primitive Datentypen werden direkt umgewandelt.

Syntax

```
<jsp:getProperty name="Bean-Instanzname" property="Attributname" />
```

Name	**Bedeutung**
name	Name der Bean-Instanz, deren Eigenschaft interessiert. Muss dem Wert von *id* in *jsp:useBean* entsprechen
property	Name der Eigenschaft, die abgefragt werden soll

10.3 Die Aktion <jsp:setProperty>

Die *<jsp:setProperty>* Aktion kann die Eigenschaften einer JavaBean verändern. Dabei werden die Setter-Methoden der JavaBean verwendet. Die Deklaration mit *<jsp:useBean>* muss immer vorrausgehen.

Es gibt verschiedene Formen setProperty zu benutzen.

1. Es wird direkt der Wert der Property mit dem Paramter value übergeben.

```
<jsp:setProperty name="user" property="name" value="klausimausi" />
```

2. Der Wert der Property wird aus den Paramtern der Anfrage (request) gelesen

```
<jsp:setProperty name="user" property="user" param="username" />
```

3. Mehrere Properties auf einmal aus der Anfrage (request) lesen

```
<jsp:setProperty name="user" property="*"/>
```

Es wird über alle Parameter der Anfrage iteriert. Entspricht der Parametername einer Anfrage einer Property so wird diese gesetzt. Besteht der Wert einer Variable der Anfrage aus einem Leerstring so wird die Proporty nicht gesetzt.

Attribut	Beschreibung
name	Name der Bean-Instanz, deren Eigenschaft bestimmt werden soll. Muss dem Wert von id in <jsp:useBean> entsprechen
property	Name der Eigenschaft, die gesetzt werden soll oder ein Wildcard für alle Eigenschaften
param	Name eines Anfrage-Parameters
value	Attributwert, der zugeordnet werden soll

Beachte, daß die Attribute *param* und *value* nie gemeinsam benutzt werden.


Given a design goal, create a code snippet using the following standard actions: <jsp:include>, <jsp:forward>, and <jsp:param>.

10.4 Die Aktion <jsp:forward>

Diese Aktion leitet die Anfrage an eine andere Webressource innerhalb derselben Webanwendung weiter. Die Ausführung der aufrufenden Seite wird sofort beendet - Anweisungen dahinter werden ignoriert - und die Kontrolle geht auf die aufgerufene Ressource über. Schon gepufferte Daten werden vorher gelöscht. Wurde der Puffer schon einmal geleert, erzeugt die Weiterleitung danach eine *IllegalStateException*. Dabei wird das *request*-Objekt dem Wert des *page* Attributs entsprechend angepasst, die sonstige Daten werden von der aufrufenden Seite übernommen und können in der aufgerufenen Seite abgefragt werden. Die Aktion kann im Rumpf <jsp:param> Kindelemente enthalten, um Parameter an die andere Ressource zu übergeben.

Beispiel:

```
<jsp:forward page="/servlet/login" />
<jsp:forward page="/servlet/login">
   <jsp:param name="username"  value="hans" />
</jsp:forward>
```

Attribut	Beschreibung
page	lokalisiert die einzuschliessende Ressource innerhalb der Webanwendung relativ zu akuellen Seite

10.5 Die Aktion <jsp:include>

Das Element fügt in die JSP eine statische Datei oder die Ausgabe ein, die eine ander JSP oder

ein Servlet erzeugt. Danach wird der Rest der JSP verarbeitet. Mit Hilfe von <jsp:param>-Kinderelemente im Rumpf der Aktion können Parameter an die zu übernehmende Ressource übergeben werden. Die inkludierte Ressource kann auf das *request* Objekt der inkludierten Seite zugreifen.

Beispiel:

```
<jsp:include page="kontakt.html" />
<jsp:include page="/index.html" flush="true" />
<jsp:include page="scripte/login.jsp" />
    <jsp:param name="username" value="Hansen" />
</jsp:include>
```

Attribut	Beschreibung
page	lokalisiert die einzuschliessende Ressource innerhalb der Webanwendung relativ zu akuellen Seite
flush	ist optional und hat als Default Wert false. Bei true wird vor der Übernahme der aufgelaugene Inhalt des Puffers an den Clienten geschickt und der Puffer geleert.

10.6 Die Aktion <jsp:param>

Die Aktionen *<jsp:include>*,*<jsp:forward>* und *<jsp:params* verwenden *<jsp:param>* als Kindelement für die Übergabe von zusätzlichen Paramtern verwendet werden. *<jsp:params>* wird in den Objectives nicht erwähnt. Es fasst mehrere *<jsp:param>* zusammen.

Beispiel:

```
<jsp:include page="kopf.jsp">
   <jsp:param name="thema" value="Sommer" />
   <jsp:param name="farbe" value="rot" />
</jsp:include>
```

11 Tag Libraries

11.1 Deklaration einer TagLib in JSP

For a custom tag library or a library of Tag Files, create the 'taglib' directive for a JSP page.

TagLib Direktiven wurden im Kapitel über Direktiven ausführlich behandelt.

Hier die Syntax zur Deklaration einer TagLib Direktive

```
<%@ taglib prefix="prefix" [uri="taglibURI" | tagdir="contextRelativePath"]%>
```

Attribut	Beschreibung
prefix	Der Prefix mappt eine Aktion zu einer TagLib Library

uri	Entweder ein eindeutiges Symbol das eine TLD spezifiziert oder ein relativer Pfad zu einer TLD Datei bzw Jar File
tagdir	Kontextrelativer Pfad zu einem Ordner mit Tag-Dateien. Der Pfad startet mit /WEBINF/tags

Beispiele:

```
<%@ taglib prefix="ora" uri="orataglib" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="mylib" tagdir="/WEB-INF/tags/mylib" %>
```

11.2 Finden der richtigen TagStruktur

Given a design goal, create the custom tag structure in a JSP page to support that goal.

Für diese Objective kann man sich nur schwer vorbereiten. Hilfreich ist sicherlich das Studium der existierenden Library, wie die Core Standard Library.

So zeigt die Aktion <*c:choose*> <*c:when*> <*c:otherwise*> wie man durch Verschachtelungen von Tags komplexe Strukturen aufbauen kann.

Einem Tag kann man Attribute mitgeben, deren Werte je nach TLD Datei fest sind oder durch Auswertung eines Ausdrucks ausgewertet werden. (siehe z.B. die Aktion <*c:if*>).

Das Ergebnis der Auswertung des Body eines Tags wird zur Ausgabe verwendet. Je nachdem wie man den Tag Handler schliesslich programmiert, wird die Auswertung des Bodys unterdrückt, (siehe Aktion <*c:if*>) oder mehrfach ausgewertet (siehe Aktion <*c:forEach*>)).

Beachte dass Ergebnis der Auswertung des Bodys, für die nächste Iteration wieder verwendet werden kann. Mehr dazu im Kapitel über die Programmierung der Tag Libraries.

11.3 Die Core Standard Tag Library

Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.

11.3.1 Überblick

Die Core Library teil sich in 3 Bereiche auf:

1. Allgemeinen Aktionen
2. Bedingungen und Schleifen
3. URL-bezogene Aktionen

Allgemeine Aktionen

Core-Tags	Bedeutung
<c:out>	Ausgabe von Daten in die Antwort
<c:set>	Setzt Variable in einen Scope
<c:remove>	löscht Daten aus einem Scope
<c:catch>	Abfangen von Fehlern

Bedingungen und Schleifen

Core-Tags	Bedeutung
<c:if>	Bedingungen
<c:choose>	Auswahl
<c:when>	Alternativen in Auswahl
<c:otherwise>	Letzte Alternative in einer Auswahl
<c:forEach>	Iterationen über eine Sammlung von Elementen
<c:forTokens>	Iterationen über Token, die durch Delimiter getrennt sind

URL-bezogene Aktionen

Core-Tags	Bedeutung
<c:import>	Inklusion von Texten
<c:url>	Umschreiben von URLs
<c:redirect>	Umleitung zu anderen Seiten
<c:param>	Übergabe von Parametern

11.3.2 Allgemeine Aktionen

Die Aktion <c:out>

Die <c:out> Aktion wertet einen Ausdruck aus und übergibt das Ergebnis an das aktuelle JspWriter-Objekt.

Da EL-Ausdrücke direkt innerhalb von Template-Daten eingefügt werden dürfen, wird diese Funktion selten verwendet, und ist vor allem mit den Attributen interessant.

Attribut	Java Typ	Dynamik	default-Wert	Beschreibung
value	Object	Ja	siehe Attribut default	Ausdruck der ausgewertet wird
escapeXml	boolean	Ja	true	Bei true werden xml Sonderzeichen in die XML Entity Codes übersetzt

default	Object	Ja	--	Ausdruck der ausgegeben wird falls die Auswertung von value null ergibt

Die Beispiele zeigen, dass der default Wert als Attribut, oder als Body angegeben werden kann.

```
<c:out value="${param.phone}" default="No Phone" />

<c:out value="${user.imageURL}">
    <c:url value="/defaultUserImage.jpg" />
</c:out>
```

Die Aktion <c:set>

Diese Aktion setzt Variablen in einem bestimmten Scope, oder setzt eine Property einer Bean.

Attribut	Java Typ	Dynamik	Beschreibung
value	String	Ja	Ausdruck der ausgewertetund als Wert genommen wird
var	String	Nein	Name der Variablen die die neue URL enthält
scope	String	Nein	Scope in der var abgelegt wird, Standard ist page
target	Object	Ja	Zielobjekt dem eine Property zugewiesen wird. Eine Bean oder Map
property	String	Ja	Property von target

```
<%-- Beispiel: Setzen einer Variable ohne Body --%>

<c:set value="${user.name}" var="Name" />

<%-- Beispiel: Setzen einer Variable mit Body --%>

<c:set  var="Name" scope="session" >
    ${user.name}
</c:set>

<%-- Beispiel: Property einer Bean setzen --%>

<c:set value="${user.name}" target="${userBeanId}" property="Name" />

<%-- Beispiel: Property einer Bean setzen --%>

<c:set target="${userBeanId}" property="Name"
     ${user.name}
</c:set>
```

Die Aktion <c:remove>

Diese Aktion entfernt eine Variable aus einem Gültigkeitsbereich

Attribut	Java Typ	Dynamik	Beschreibung
var	String	Nein	Name der Variable, die entfernt werden soll
scope	String	Nein	Scope in der var abgelegt wird, Standard ist page

Beispiel:

```
<c:remove var="loggedOut" scope="session" />
```

Die Aktion <c:catch>

Diese Aktion fängt Fehler ab, es ist möglich die ausgelöste Exception an eine Variable im PageScope zuzuordnen.

```
<c:catch var="importException">
  <c:if test="${importException != null}">
      <jsp:forward page="input.jsp">
          <jsp:param name="msg" value="Invalidate date format" />
      </jsp:forward>
  </c:if>
</c:catch>
```

Die Aktion <c:catch> hat nur ein einziges optionales Attribut *var*. Diese Variable liegt automatisch im pageScope.

11.3.3 Bedingungen und Schleifen

Die Aktion <c:if>

Die Auswertung des Bodys eine Aktion *<c:if>* erfolgt nur dann, falls die der im Attribut *test* angegebene Ausdruck *true* ergibt.

```
<c:if test="booleanExpression">
   JSPElements
</c:if>
```

Attribut	Java Typ	Dynamik	Beschreibung
test	boolean	Ja	Ausdruck der ausgewertet wird
var	String	Nein	Variablenname
scope	String	Nein	Scope von var

Statt Auswertung eines Bodys kann man auch das Ergebnis der Auswertung in einer Variablen abspeichern.

```
<c:if test="booleanExpression" var="var" scope="page|request|session|application" />
```

Die Aktionen <c:choose> <c:when> <c:otherwise>

Diese Aktionen entspricht dem Java *switch()* Statement. Die Aktionen *<c:when>* und *<c:otherwise>* sind Kindelemente der Aktion*<c:choose>*. Das Vorhandensein einer Aktion *<c:otherwise>* ist optional.

```
<c:choose>
   <c:when test="${product.onSale}">
       ${product.salesPrice} On sale !
   </c:when>
   <c:otherwise>
       ${product.price}
   </c:otherwise>
</c:choose>
```

Die Aktion *<c:when>* ist die einzigste die ein Attribut besitzt. Ergibt die Auswertung des Ausdruck vom *test* den Wert *true* so wird der Body ausgewertet. Innerhalb der Aktion *cc:choose>* können mehrere *<c:when>* Ausdrücke sein. Es wird der Body der ersten *<c:when>* Aktion ausgeführt, die true ergibt. Ansonsten wird der Body der *<c:otherwise>* Aktion ausgeführt.

Die Aktion <c:forEach>

Es gibt 2 Arten diese Aktion zu verwenden.

1. Iteration über eine feste Anzahl von Iterationen
2. Iteration über eine Collection

Iteration über eine feste Anzahl von Iterationen

Beispiel:

```
<%-- Iterate five times, writing 0,2,4,6 --%>
<c:forEach begin="0" end="6" step="2" var="current">
   ${current}
</c:forEach>
```

Das Attribut *begin* und *end* sind wenn man über eine feste Anzahl von Iterationen iteriert Pflichtattribute.

Iteration über eine Collection

Beispiel:

```
<%-- Iterate over all request parameters --%>

<c:forEach items="${param}" var="current">
   Name: ${current.key}
   Vorname: ${current.value}
</c:forEach>
```

Attribut	Java Typ	Dynamik	Beschreibung
begin	int	Ja	Index für Anfang der Iteration
end	int	Ja	Index für das Ende der Iteration
items	Collection, Iterator, Enumeration, Map oder ein Array	Ja	Darüber wird iteriert
step	int	Ja	Schrittweite der Schleife, Standard ist 1.
var	String	Nein	Variable das das aktuelle Element der Iteration hält
varStatus	String	Nein	Variable enthält interne Informationen über die Schleifenduchlauf Ist vom Interface LoopTagStatus

varStatus kann man nur innerhalb einer forEach bzw forToken Aktion verwenden. Das Objekt das varStatus hält ist vom Interface *LoopTagStatus* und hat folgende Methoden:

```
public Integer getBegin()  // begin Wert oder null
public Integer getCount()  // Aktuelle Iteration beginnend bei 1
public Object  getCurrent() // Aktuelle Variable der Iteration
public int     getIndex()  // Index aktueller Variable beginned bei 0
public Integer getStart()  // start Wert oder null
public Integer getStep()   // step Wert oder null
public boolean isFirst()   // Gibt true für den ersten Durchlauf
public boolean isLast()    // Gibt true für den letzten Durchlauf
```

Die Aktion <c:forTokens>

Diese Aktion iteriert für jeden Token innerhalb eines String. Die Token werden über sogenannte Trennzeichen (delimiters) getrennt.

Beispiel:

```
<%-- Iteriert über Tokens die durch Vertikalstrich getrennt sind --%>

<c:forTokens items="${tokens}" delims="|" var="current">
    <c:out value="${current }" />
</c:forTokens>
```

Attribut	Java Typ	Dynamik	Beschreibung
begin	int	Ja	Index für Anfang der Iteration
end	int	Ja	Index für das Ende der Iteration
items	Collection, Iterator, Enumeration, Map oder ein Array	Ja	Darüber wird iteriert
delims	String	Ja	Liste aller Trennzeichen
step	int	Ja	Schrittweite der Schleife, Standard ist 1.
var	String	Nein	Variable das das aktuelle Element der Iteration hält
varStatus	String	Nein	Variable enthält interne Informationen über die Schleifenduchlauf Ist vom Interface LoopTagStatus

11.3.4 URL-bezogene Aktionen

Die Aktion <c:url>

Diese Aktion wird für URL-Rewriting genommen. Ist der Aufruf Teil einer Sitzung, so wird die Session-ID der URL hinzugefügt.

Attribut	Java Typ	Dynamik	Beschreibung
value	String	Ja	Pflichattribut - Eine absolute oder relative URL
context	String	Ja	Optional - Falls die Umleitung zu einer anderen Applikation gehört, brauchen wir den Context Pfad der Zielapplikation
var	String	Nein	Optional - Name der Variablen die die neue URL enthält
scope	String	Nein	Optional - Scope in der var abgelegt wird, Standard ist page

Werden in der URL-Parameter mitgegeben, so können diese als Unterelemente mit der Aktion <c:param> mitgegeben werden.

```
<c:url value="product.jsp">
    <c:param name="id" value="${product.id}" />
    <c:param name="customer" value="$Rudi Ratlos" />
</c:url>
```

Die Aktion <c:import>

Im Gegensatz zur *include*-Direktive und die *<jsp:include>* Standardaktion, kann man mit der Aktion *<c:import>* auch Ressourcen aus einer anderen Webanwendung einbinden. Es können

auch Parameter mitgegeben werden.

```
<c:import var="xmldaten"
     url="http://www.einstiegjsp.de/testdaten.xml">
  <c:param name="sprache" value="${header['accept-language']}" />
</c:import>
```

Es ist auch möglich Ressourcen über ftp einzubinden.

Die eingelesene Ressource muss nicht sofort eingebunden werden, sondern kann mit dem Attribut *var* und *scope* einer Variablen zugewiesen werden.

Eine andere Möglichkeit ist es, sich über das Attribut *varReader* eine Instanz von *java.io.Reader* geben zu lassen. Diese Instanz gibt Methoden wie *read()*,*skip* usw. zur Verfügung. Ist ein *varReader* definiert, so wird im Body der *<c:import>* Aktion darauf zugegriffen.

Attribut	Java Typ	Dynamik	Beschreibung
url	String	Ja	Pflicht - Die URL der Ressource die importiert wird
context	String	Ja	Optional - Context-Path für eine Webanwendung im selben Container. Startet mit einem Slash
var	String	Nein	Optional - Variable die den Inhalt der Ressource hält
scope	String	Nein	Optional - Scope der Variable. Standard ist page
varReader	String	Nein	Optional - Variable die einen java.io.Reader hält
charEncoding	String	Ja	Optional - Zeichensatz der Ressource

Beispiel:

```
<c:import url="navigation.jsp" />

<c:import url="http://wwww.blabla.de"
    varReader="xmlSource">
    <x:parse var="doc" xml="${xmlSource}" scope="application" />
</c:import>
```

Die Aktion <c:redirect>

Diese Aktion dient zur Umleitung der aktuellen URL. Es gibt dem Browser die Antwort, er solle die Anfrage mit der neuen URL betätigen.

Attribut	Java Typ	Dynamik	Beschreibung
url	String	Ja	Pflicht - Ein relativer Pfad, oder eine absolute URL
context	String	Ja	Optional - Context-Path für eine Webanwendung im selben Container. Startet mit einem Slash

Beispiel:

```
<c:redirect url="navigation.jsp" />

<c:redirect url="navigation.jsp" >
    <c:param name="user" value="Apahachi" />
</c:redirect>
```

Die Aktion <c:param>

Dient der Paramterübergabe in den Aktionen <*c:redirect*>, <*c:import*> und <*c:url*>.

Es gibt 2 syntaktische Schreibweisen. Einmal wird der Wert des Paramters als Attribut value mitgegeben. Das andereMal wird er über den Body mitgegeben.

```
<c:param name="parameterName" value="parameterWert">

<c:param name="parameterName">
   "paramterWert"
</c:param>
```

Die Aktion <*c:param*> hat nur als Kindelement einen Sinn.

12 Aufbau eigener Tag Libraries

12.1 Aufbau von Classic Tag Libraries

Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.

12.1.1 Überblick über die Tag Interfaces

Mit Einführung der JSP 2.0 sind die Classic Tag Libraries unbedeutender geworden, da mit den Simple Tag Libraries eine einfachere Möglichkeit zur Bildung eigener Tags eingeführt werden.

Für jedes eigene Tag, muss man eine Java Klasse beschreiben, die ein bestimmtes Interface erfüllen muss. In den TLD - Dateien wird der Bezug von den in der JSP verwendeten Tags und den Java Klassen hergestellt. Je nach Komplexität der Tags wird ein anderes Interface verwendet.

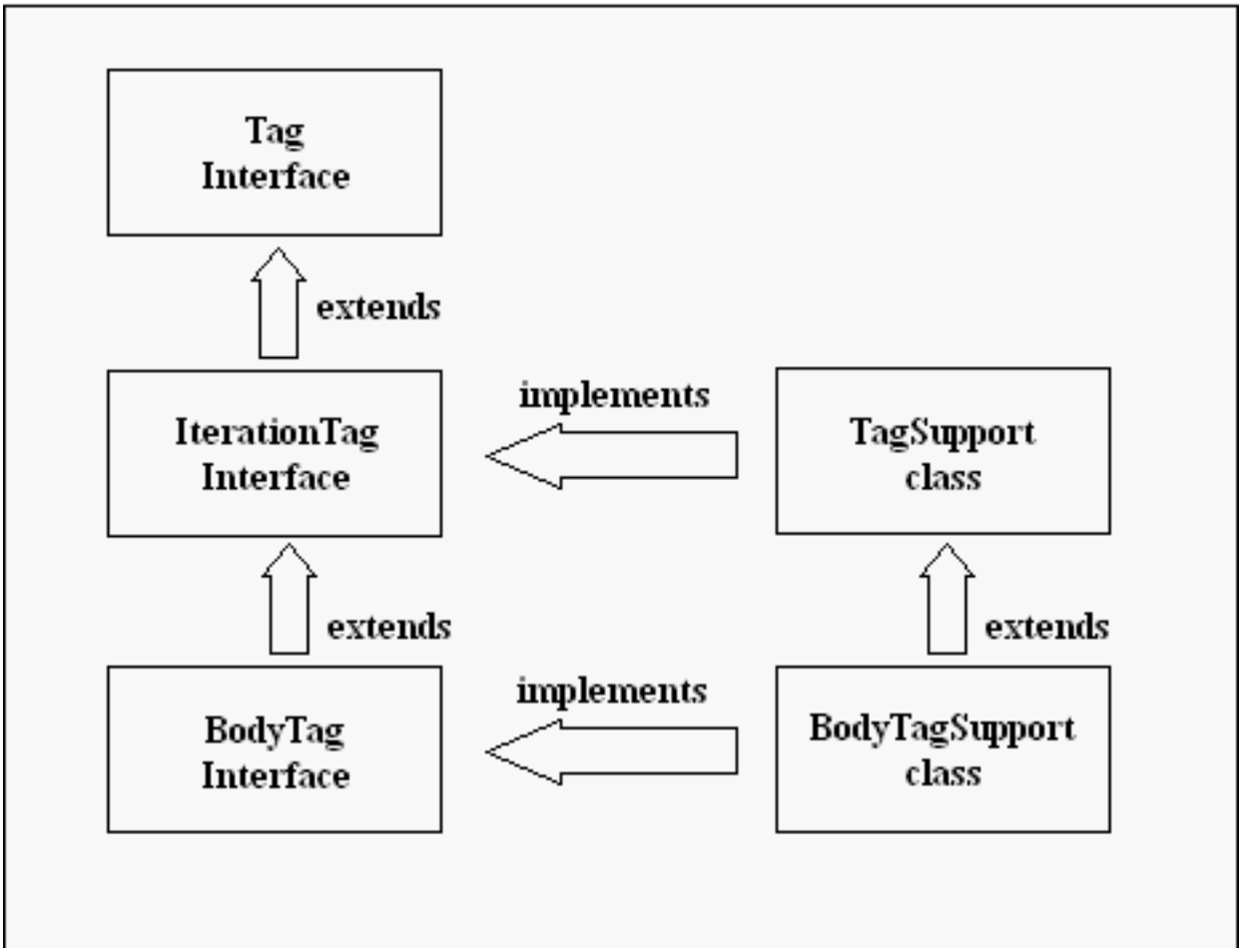


Abb. 12.1: Die Tag Interfaces

Hier ein Überblick

- Das **TagInterface** ist das einfachste Interface, und reicht für die meisten Tags aus
- Das **IterationTag** Interface erweitert das Tag Interface, um die Eigenschaft den Body einer Iteration mehrfach mehrfach auszuführen.
- Das **BodyTagInterface** erweitert das Iteration Tag Interface. Es wird immer dann genommen wenn der Body der Aktion nicht einfach an die Antwort weitergereicht, sondern während der Ausführung der Aktion bearbeitet werden soll.

Für das Iteration Tag Interface und das Body Tag Interface gibt es Standard Implementierungen. Der Programmierer kann Unterklassen dafür bilden, um nicht alle notwendigen Methoden der Interfaces neu implementieren zu können.

12.1.2 Das TagInterface

Jeder TagHandler ist eine Java Klasse und muss alle Methoden des TagInterface implementieren. Hier eine Auflistung der Methoden und Konstanten des Tag Ingerfaces:

Methode	**Beschreibung**
int doStartTag()	Wird nach dem Lesen des StartTags ausgeführt
int doEndTag()	Wird nach dem Lesen des EndTags ausgeführt
Tag getParent()	Gibt den TagHandler des direkt Eltern Tags zurück

void release()	Wird vom TagHandler aufgerufen, wenn der Container den Handler nicht mehr benötigt
void setPageContext (PageContext)	Dies wird nach dem Instantiieren eines TagHandlers aufgerufen Mit dem PageContext hat man Zugriff auf alle impliziten Objekten, und den Variablen in den Scopes
void setParent(Tag)	Wird bei verschachtelten Tags verwendet, um auf den Handler des übergeordneten Tags zuzugreifen

Konstante	**Beschreibung**
EVAL_BODY_INCLUDE	Rückgabewert für das doStartTag() Der Body des Tags wird ausgewertet und in den Ausgabe eingefügt
SKIP_BODY	Rückgabewert für das doStartTag() Der Body des Tags wird nicht ausgewertet
EVAL_PAGE	Rückgabewert für das doEndTag() Nach Auswertung des Tags wird der Rest der JSP Seite auch ausgewertet
SKIP_PAGE	Rückgabewert für das doEndTag() Verhindert die Auswertung der Rest der JSP Seite

Die folgende Abbildung zeigt, wann welche Methode bei der Bearbeitung eines Tags aufgerufen wird:

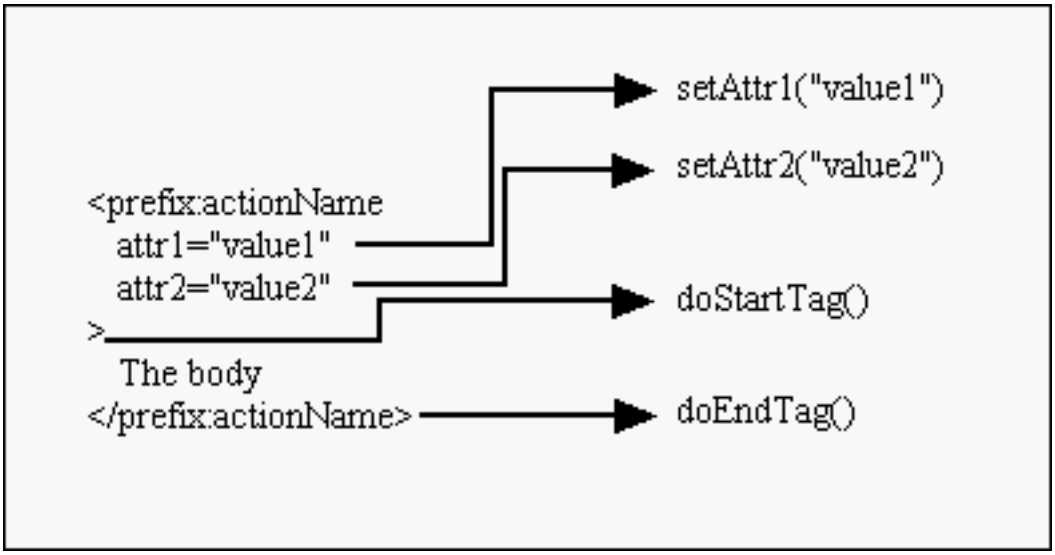


Abb. 12.2: Events beim Tag Interface

Entscheident sind die die Methoden *doStartTag()* und *doEndTag()*. Beide Methoden können eine *JspException* werfen. Die Methode *doStartTag()* wird u.a. zum initialisieren des TagHandlers verwendet. Sie sollte eine von 2 Integerkonstanten zurückgeben. Ist der Rückgabewert der Integer **Tag.EVAL_BODY_INCLUDE** so wird der Body der Aktion ausgewertet und in die Ausgabe eingefügt. Ist der Rückgabewert der Integer **Tag.SKIP_BODY** wird der Body nicht ausgewertet. Dies wird zum Beispiel bei der Implementierung eines *if* Tags benutzt.

Die Methode *doEndTag()* hat auch 2 mögliche Rückgabewerte. Bei *Tag.EVAL_PAGE* wird der

Rest der JSP Seite ausgewertet. Ansonsten bei Rückgabewert *Tag.SKIP_PAGE* wird die weitere Auswertung der JSP Seite ignoriert.

Das folgende Bild fasst den Kontrollfluss bei Auswertung eines Tags zusammen:

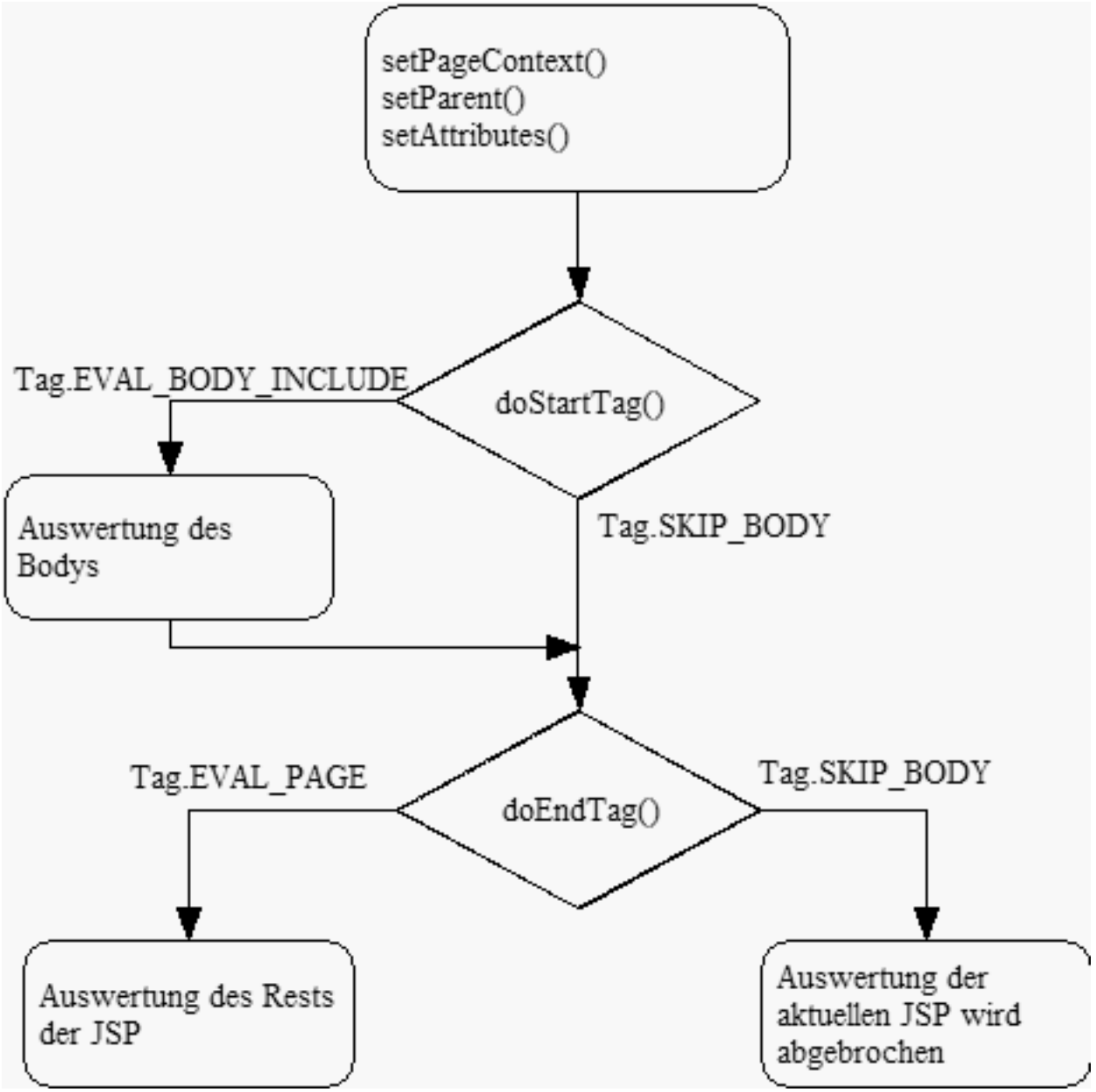


Abb. 12.3: Flusskontrolle bei Abarbeitung eines Tag Interfaces

12.1.3 Das Iteration Tag Interface

Das IterationTag Interface erweitert das Tag Interface, um nur eine Methode und einer Konstante

Methode / Konstante	**Beschreibung**
int doAfterBody()	Dies Methode wird nach der Ausführung des Bodys aufgerufen. Als möglicher Rückgabewert ist entweder der Integer IterationTag.EVAL_BODY_AGAIN oder Tag.SKIP_BODY
EVAL_BODY_AGAIN	Möglicher Rückgabewert von doAfterBody(). Bei diesem Rückgabewert wird die Auswertung des Bodys wiederholt

Der Methodenfluss wird duch die folgende Abbildung deutlich:

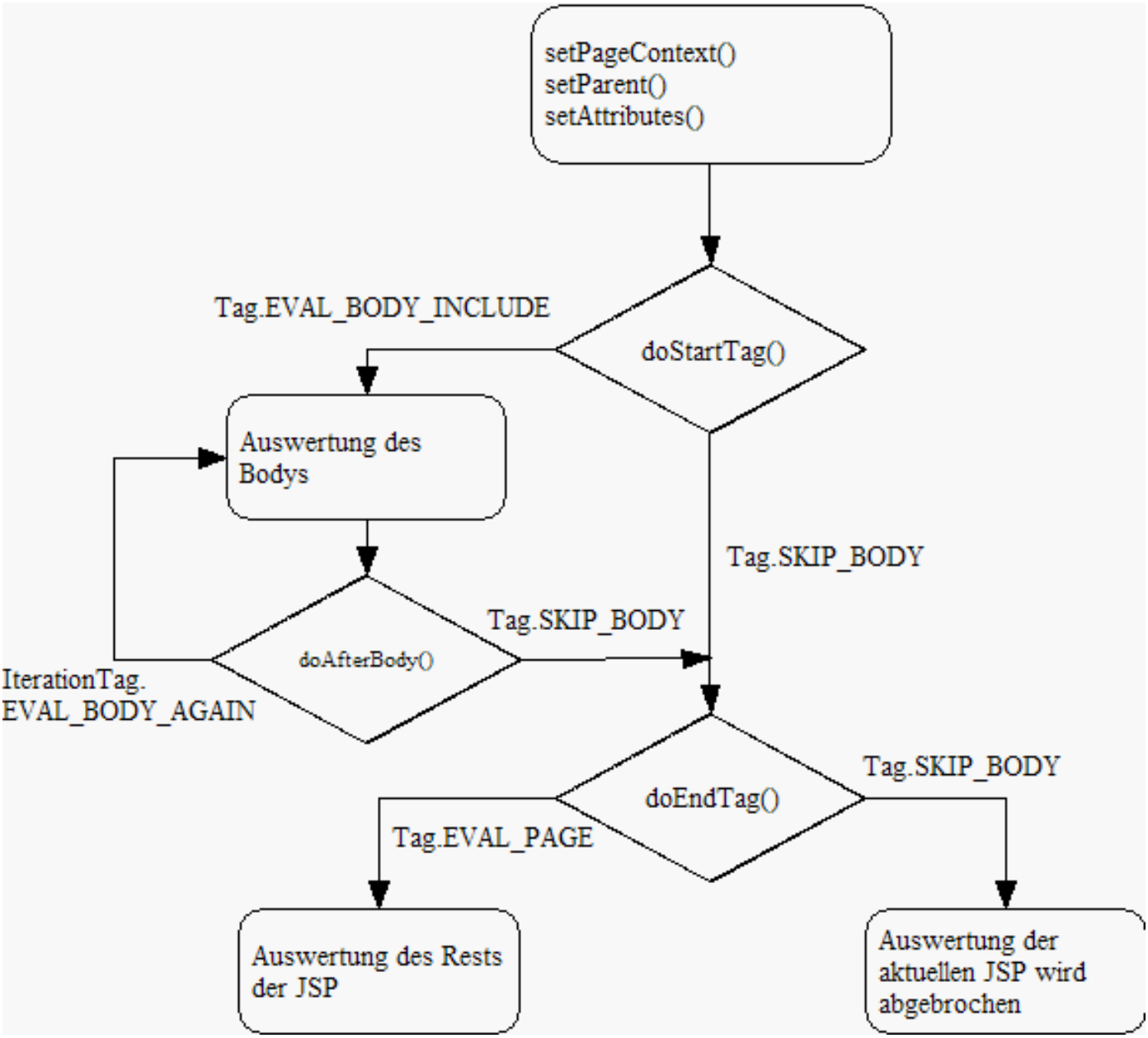


Abb. 12.4: Flusskontrolle bei Abarbeitung eines IterationTag Interfaces

Beispiel:

Zu entwickeln ist ein Tag loop das den Body mehrfach entsprechend der Variable count mehrfach ausgeführt wird.

```
<%@ taglib prefix="test" uri="/WEB-INF/sampleLib.tld" %>
<html><body>
    <test:loop count="5" >
        Hello World!<br>
    </test:loop>
</body></html>
```

Es muss eine Klasse mit dem IterationTag Interface implementiert werden. Die Klasse soll LoopTag heissen. Die Zuordnung des Tags <test:loop> zur Klasse LoopTag wird weiter unten erläutert.

```
package sampleLib;

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

// Beachte dass TagSupport das Interface IterationTag implementiert.
// Von der Klasse TagSupport werden u.a. die Methoden int doEndTag(),
// setParent(), getParent(), setPageContext(), release() beerbt

public class LoopTag extends TagSupport
{
    // Für alle Attribute muss eine setter Methode definiert werden

    private int count = 0;

    public void setCount(int count){
        this.count = count;
    }

    public int doStartTag() throws JspExpression{
        if (count>0)
            return EVAL_BODY_INCLUDE;
        else
            return SKIP_BODY;
    }

    public int doAfterBody() throws JspExpression{
        if (--count > 0)
            return EVAL_BODY_AGAIN;
        else
            return SIP_BODY;
    }

    // Die Methode doEndTag() muss in diesem Fall nicht
    // reimplementiert werden, da sie standardmaessig von der
    // Oberklasse TagSupport den Integer EVAL_PAGE zurückgeben
}
```

Die Zuordnung des tags <test:loop> zur Klasse LoopTag muss in der TLD-Datei definiert werden:

```
<tag>
    <name>loop</name>
    <tag-class>sampleLib.LoopTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>count</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

12.1.4 Das BodyTag Interface

Das BodyTag Interface erweitert das IterationTag Interface. Es ermöglicht den Body, bevor er zur Ausgabe gesendet wird, zu bearbeiten.

Das BodyTag Interface erweitert das IterationTag um folgende Methoden und Konstanten:

Methode / Konstante	**Beschreibung**
void setBodyContent()	Setzt eine Instanz der Klasse BodyContent.BodyContent enthält den Inhalt des Bodys, und hat Methoden um diesen zuzugreifen.
void doInitBody()	Dies erlaubt bevor der Body bearbeitet wird, den BodyContent zu bearbeiten. Diese Methode wird direkt nach setBodyContent() aufgerufen. Diese Methode wird häufig reimplementiert.
EVAL_BODY_BUFFERED	Möglicher Rückgabewert von doStartTag() und doAfterBody(). Der Inhalt der Auswertung des Bodys wird gepuffert. Die Auswertung des Bodys wird nochmals wiederholt, allerdings mit dem Body der der Auswertung der

	letzten Iteration entspricht

Das folgende Bild zeigt, wann die Methoden *setBodyContent()* und *doInitBody()* vom Container ausgelöst werden.

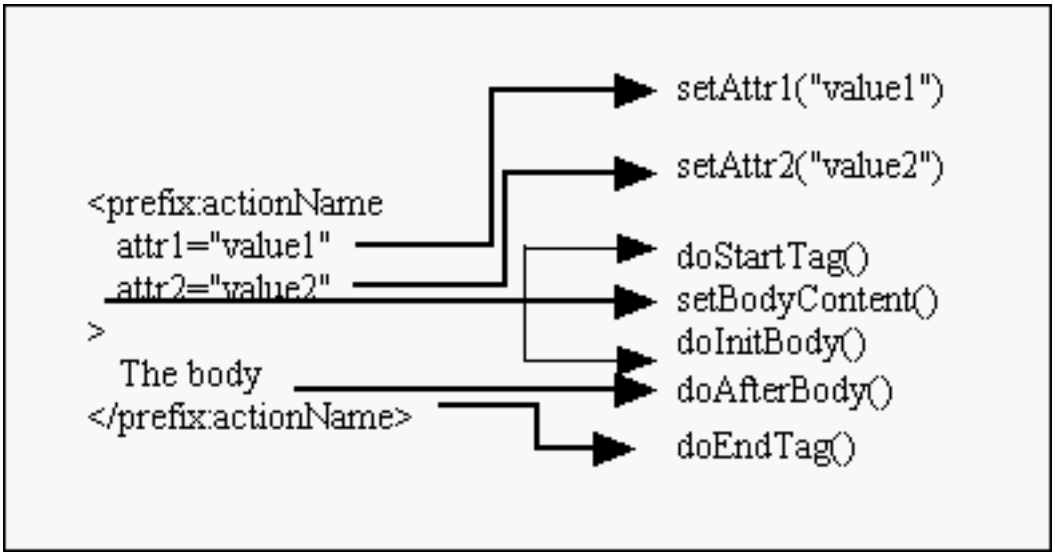


Abb. 12.5: Events beim BodyTag Interface

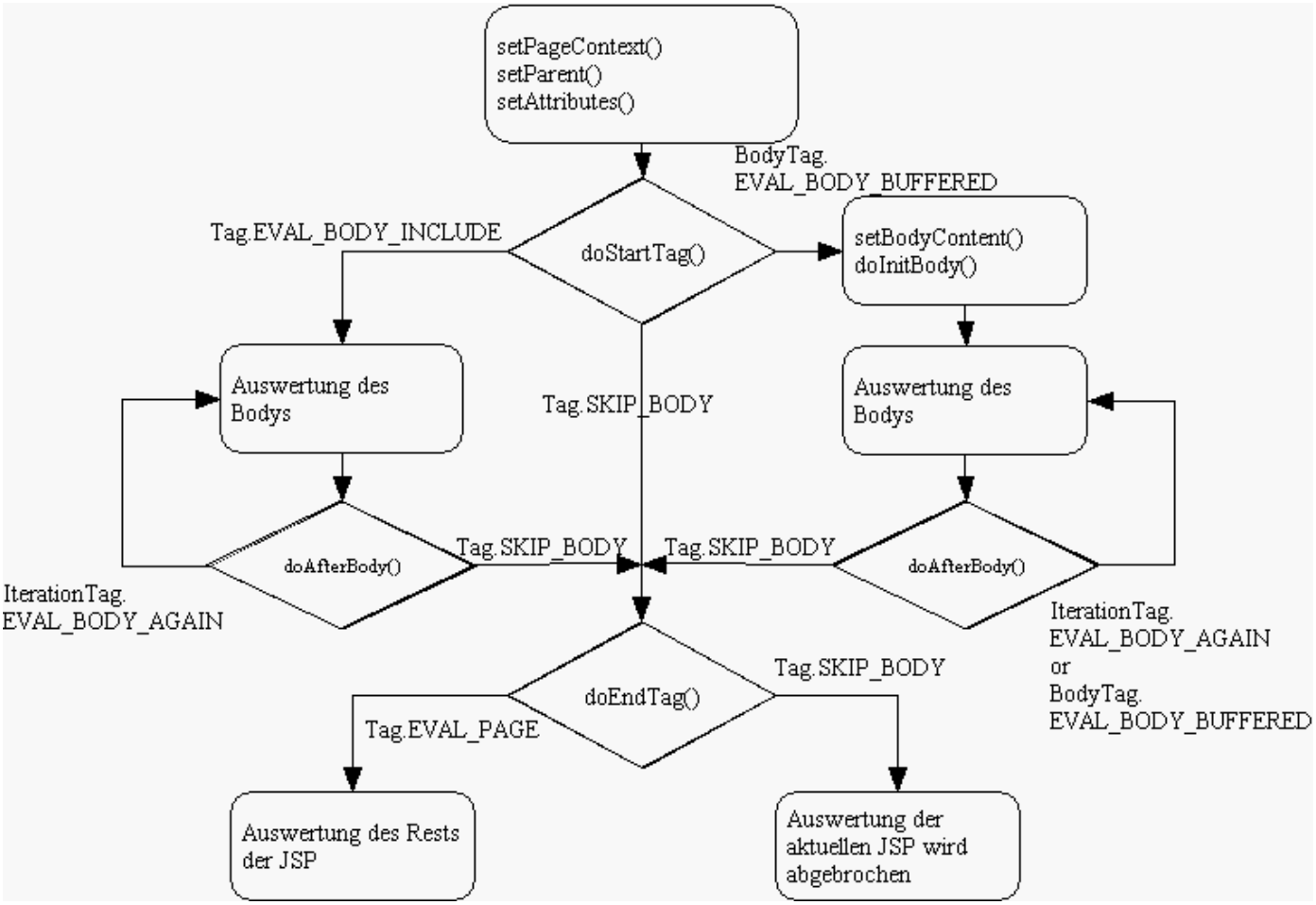


Abb. 12.6: Flussdiagramm BodyTag Interface

Als konkrete Implementation des BodyTag Interfaces wird meistens eine Unterklasse von der

Klasse BodyTagSupport gebildet.

Die Klasse BodyTagSupport bietet u.a. folgende Methoden:

setBodyContent(BodyContent)	**Setzt ein BodyContent Objekt**
BodyContent getBodyContent()	Gibt den BodyContent des Tags zurück
JSPWriter getPreviousOut()	Gibt einen Writer zurück, um den Body zu bearbeiten. Dies entspricht bodyContent.getEnclosingWriter(). Der Inhalt dieses Writers wird nicht sofort ausgegeben, sondern erstmal für die Weiterverarbeitung (in der nächsten Iteration) gepuffert

12.2 Die PageContext API

Using the PageContext API, write tag handler code to access the JSP implicit variables and access web application attributes.

PageContext ist ein Objekt, auf das man als implizite Variable mit dem Ausdruck ${pageContext} zugreifen kann.

PageContext bietet:

- eine API um Attribute in den verschiedenen Gültigkeitsbereichen (Scopes) zu lesen bzw. schreiben
- einfacher Zugriff auf öffentliche Variablen wie out,session,page,servletConfig bzw servletContext
- Ein Meachanismus um die Anfrage an weitere aktive Komponenten mit forward bzw include weiterzuleiten
- Ein Mechanismus zur Fehlerbehandlung

Hier die wichtigsten Methoden die PageContext anbietet:

```
public abstract class JspContext {

        public abstract void setAttribute(String name, Object value); // setzt Attribut im page Scope
        public abstract void setAttribute(String name, Object value, int scope);
        public abstract Object getAttribute(String name); // liest Attribut aus PageScope
        public abstract Object getAttribute(String name, int scope);
        public abstract Object findAttribute(String name); //sucht in allen Scopes
        public abstract void removeAttribute(String name); // löscht im PageScope
        public abstract void removeAttribute(String name, int scope);
        public abstract int getAttributesScope(String name); // Gibt scope zurück indem Attribut gefunden wurde
        public abstract Enumeration getAttributeNamesInScope(int scope);
        public abstract JspWriter getOut(); // gibt den JSP Writer der Seite zurück

}
```

```
public abstract class PageContext extends JspContext {

        public abstract javax.servlet.http.HttpSession getSession();
        public abstract java.lang.Object getPage();
        public abstract javax.servlet.ServletRequest getRequest();
        public abstract javax.servlet.ServletResponse getResponse();
        public abstract java.lang.Exception getException();
        public abstract javax.servlet.ServletConfig getServletConfig();
        public abstract javax.servlet.ServletContext getServletContext();
```

```
        public abstract void forward(java.lang.String relativeUrlPath)
                throws javax.servlet.ServletException, java.io.IOException;
        public abstract void include(java.lang.String relativeUrlPath)
                throws javax.servlet.ServletException, java.io.IOException;
        public abstract void handlePageException(Throwable e)  // für unbehandelte Ausnahmefälle
                throws javax.servlet.ServletException, java.io.IOException;
}
```

Für die verschiedenen Scopes sind in PageContext Konstanten definiert. Sie lauten

```
public static final int APPLICATION_SCOPE
public static final int PAGE_SCOPE
public static final int REQUEST_SCOPE
public static final int SESSION_SCOPE
```

12.3 Umgang zur Programmierung verschachtelter Tags

Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.

Es gibt 2 Methoden um von einem TagHandler eines inneren Tags, auf den TagHandler eines äusseren Tags zuzugreifen.

Methode	**Beschreibung**
Tag setParent()	Dies gibt des TagHandler der übergeordneten Klasse zurück. Existiert kein übergeordneter TagHandler so wird der Wert *null* zurückgegeben
TagSupport. findAncestorWithClass(tag,class)	Dies statische Methode wird angenommen wenn man nicht die Verschachtelungshierarchie der Tags kennt. Das erste Argument ist die Instanz eines TagHandlers von der gesucht wird. Das zweite Argument ist eine Klasse oder Interface. Die erste Instanz die gefunden wird, die von der Klasse oder Interface ist, wird zurückgegegen.

Hier ein Beispiel zur Verwendung findAncestorWithClass

```
public class SQLParamTag extends SimpleTagSupport {

...

   public void doTag() throws JspException {
       SQLExecutionTag parent = (SQLExecutionTag)
                 find AncestorWithClass(this,SQLExecution.class);
       if (parent == null) {
           throw new JspTagException("The param action is not " +
                    "enclosed by a supported action type");
       }
       parent.addSQLParameter(value);
   }
}
```

Bei SimpleTags erfüllt ein TagHandler das Interface *SimpleTag* und wird meist als Unterklasse von *SimpleTagSupport* implementiert. Beim einem klassischen TagHandler wird i.a. eine Unterklasse von *TagSupport* implementiert. Beide Klassen verstehen besitzen die Methode:

12.4 Aufbau von Simple Classic Tag Libraries

Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.

12.4.1 Vergleich mit den klassischen Tag Interfaces

Das SimpleTag Interface ist seit JSP 2.0 eingeführt, und vereinfacht die Programmierung von eigengeschreibenen Tags.

Es hat folgende Vorteile:

- einfacheres Interface
- es ermöglicht beliebig komplexe Tags zu entwickeln
- einheitliches Interface, egal für welche Art von Tags

Es hat auch ein paar Nachteile:

- Erst gültig ab JSP 2.0 also mit J2EE 1.4
- es können keine Skriplets im Body sein
- Es wird bei jedem Aufruf eine neue Instanz des TagHandlers erzeugt. Man kann denselben TagHandler nicht cachen.

12.4.2 Das SimpleTag Interface

Der Programmierer schreibt Klassen, die das SimpleTag Interface erfüllen. Meist entwickelt er eine Unterklasse der abstrakten Klasse *SimpleTagSupport*. Diese Unterklasse überschreibt dann meistens nur die *doTag()* Methode.

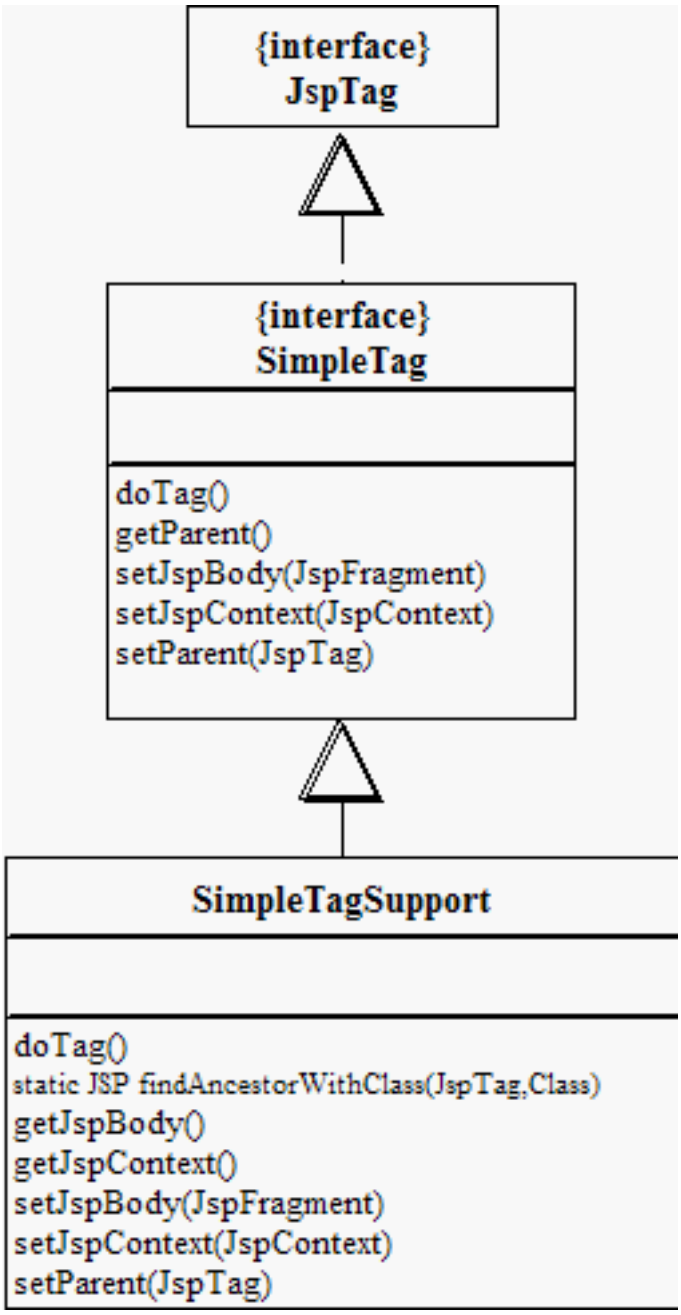


Abb. 12.7: Benutzung SimpleTagSupport Klasse

Während bei den Classic Custom Tags es verschiedene Arten von Events gibt (*doStartTag()*, *doEndTag()* und *doAfterBody()* wird bei dieser Version nur der doTag() bei der Bearbeitung des Bodys ausgelöst.

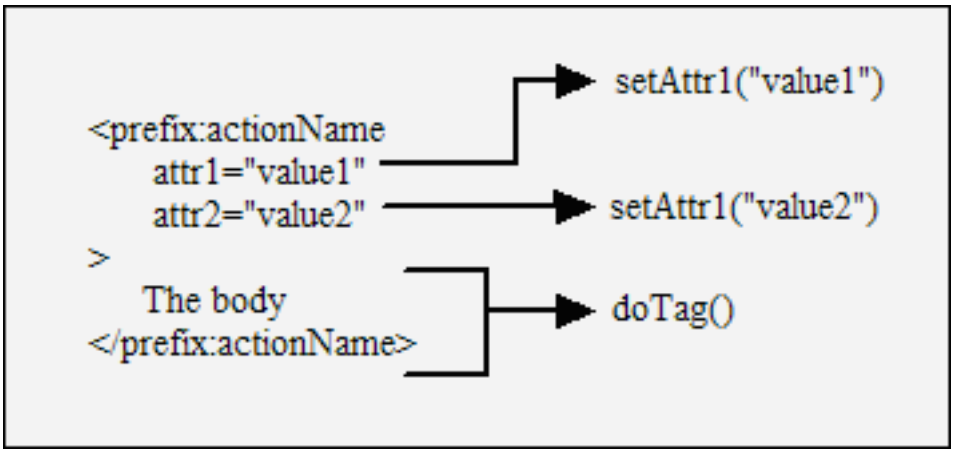


Abb. 12.8: Events bei Bearbeitung mit SimpleTag

I.a. wird der Programmierer eine Unterklasse von *SimpleTagSupport* schreiben. In dieser Unterklasse schreibt er für jedes übergebene Attribut eine entsprechende setter-Methode und überschreibt die doTag() Methode.

Die Klasse *SimpleTagSupport* bietet einige Methoden, die die Implementierung des Tags erleichtern.

- public void doTag() throws JspExeption, SkipPageException, java.io.Exception

Dies ist i.a. die einzigste Methode die der Programmierer überschreibt, und wird beim Lesen des Bodys des Tags aufgerufen. Mit dem Werfen einer *SkipPageException* wird veranlasst dass der Rest der JSP Seite nicht weiter bearbeitet wird (entspricht TAG.SKIP_BODY Konstante, bei den Classic Custom Tag Interfaces).

- public JspTag getParent()

Dies gibt den TagHandler des übergeordneten Tags zurück. JspTag ist das Interface das alle TagHandler erfüllen, hat aber keine eigenen Methoden. D.h der Programmierer muss nach Anwendung der getParent() Methode das erhaltene Objekt auf die konkrete TagHandler Klasse casten. Hat der Tag keinen übergeordneten Tag, so wird null zurückgegeben

- public void setParent(JspTag parent)

Diese Methode wird meistens nur vom Container verwendet.

- public static final JspTag findAncestorWithClass(JspTag from, Class klass)

Findet den nächsten Vater des Tags from, der der Klasse klass entspricht. Intern wird die getParent() Methode verwendet.

- public JspFragment getJspBody()

Enthält den Body des Tags in Form eines JspFragment Tags. Die Verwendung wird weiter unten erklärt

- public void setJspBody(JSPFragment body)

Diese Methode wird meistens nur vom Container verwendet.

- public JspContext getJspContext()

Dies ist eine abstrakte Klasse. Die Klasse PageContext ist i.a. die einzigste konkrete Unterklasse von JspContext. JspContext dient zum Zugriff auf die Attribute der Scopes, und mit getOut() erhält man die aktuelle Ausgabe. Diese Methode wird sehr häufig angewendet.

- public void set JspContext setJspContext(JspContext context)

Diese Methode wird vom Container verwendet, und selten vom Programmierer direkt benutzt.

12.4.3 Beispiel für die Verwendung eines SimpleTag Interfaces

Es wird bewusst ein Beispiel mit einem Tag ohne Body genommen. Der Tag hat folgenden Aufbau

```
<test:gruesse />
<test:gruesse name='Klaus' />
<test:gruesse name='Meucht' sex='m' />
<test:gruesse name='Meucht' sex='f' />
```

Es wird ausgegeben:

```
Hallo
Hallo Klaus
Hallo Herr Meucht
Hallo Frau Meucht
```

Der TagHandler wird folgendermassen programmiert

```
public class GrussTag extends SimpleTagSupport(){

   private String name,sex ;

// Für jedes Attribut benoetigt eine setter Methode


   public void setName(String name){
      this.name = name;
   }

   public void setSex(String mOrf){
      this.sex = mOrf;
   }

// die doTag() Methode ist die einzigste Event-Methode mit SimpleTag Interfaces

   public void doTag() throws JspException, IOException{
         PageContext pageContext = (PageContext) getJspContext();
         HttpServletResponse resp = ( HttpServletResponse) pageContext.getResponse();
         PrintWriter pw = resp.getWriter();

    // Alternative JspWriter pw = getJspContext().getOut();
    // JspWriter funktioniert wie PrintWriter, kann zusaetzlich Daten zwischenspeichern.

         if (name == null){
             pw.println("Hallo");
             return ;
         }
         if (sex == null){
             pw.prinln("Hallo " + name );
             return ;
         }

         if ( sex.compareToIngoreCase("m")== 0 ){
             pw.println("Hallo Herr " + name );
         }else{
             pw.println("Hallo Frau " + name );
         }
   }

}
```

12.4.4 Zugriff auf den Body des Tags

Ein TagHandler der das SimpleTag Interface erfüllt, getJspBody() und sollte ein JspFragment Objekt zurückgeben. Das JspFragment kapselt den Body und hat nur 2 Methoden.

- getJspContext()
- invoke(Writer)

Die Methode *invoke(Writer)* schreibt die Auswertung des Body in das Argument Writer. Ist der Wert des Arguments null so wo wird der Body, in den Writer, der sich durch *getJspContext().getOUt()* ergibt, ausgegeben. In diesem Fall wird der Body in die normale Ausgabe geleitet. Immer dann wenn den Body selbst nicht bearbeiten will, übergibt man den null Wert als Argument der invoke() Methode.

Immer dann wenn man den Body bearbeiten will, kann man ein eigenes Writer Objekt erzeugen, und darin die Ausgabe puffern, und dann in die Ausgabe zurückgeben.

```
...
StringWriter evalResult = new StringWriter();
StringBuffer buffer = evalResult.getBuffer();
body.invoke(evalResult);
// Die Auswertung des Body steht nun in buffer
bearbeite(buffer);
getJspContext().getOut().print(buffer);
...
```

12.4.5 Schleifen mit dem SimpleTag Interface

Schleifen werden intern in der doTag() Methode implementiert. Ein einfaches Beispiel verdeutlicht dies.

```
<test:forTokens items="${tokens}" var="current" >
    <c:out value="${current}" />
</test:forTokens >
```

Der entsprechende TagHandler kann folgendermassen implementiert werden:

```
// imports werden weggelassen

puclic class ForTokenTagHandler extends SimpleTagSupport{
    private Collection items ;
    private String var ;

    public setItems(Collection items){
        this.items = items ;
    }

    public setVar(String var){
        this.var = var ;
    }

    public void doTag() throws JspException,IOException {
        JspFragment body = getJspBody();
        if (body != null){
              Iterator iter = items.getIterator();
              while (iter.hasNext()){
                  Object currentValue = iter.next();
                  getJspContext().setAttribute(var, currentValue);
                  body.invoke(null);
              }
        }
    }
}
```

Beachte:

- Die Iteration wird in der doTag() Methode abgehandelt.
- In der doTag() Methode wird am Beginn der Iteration, ein Attribut im PageScope gesetzt. Das Attribut hat als key den Wert der Variable var (im Beispiel "current"). Der Wert des Attribut ist das aktuelle Token in der Iteration.

- Bei der Auswertung des Bodys, durch die Methode invoke(null), wird der Wert des Attributs "current" ausgegeben.

12.5 Das Tag File Model

Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.

12.5.1 Tag Dateien

Es gibt seit JSP 2.0 ein Verfahren ohne Java-Kenntnisse eigene Tags für JSP zu entwickeln. Dem Tag werden keine Java Klassen zugeordnet, sondern direkt JSP Codefragmente.

Diese Codefragmente werden in sogenannten **.tag** Dateien abgelegt. Sind diese Tag-Dateien in XML Syntax geschrieben, so wird die Endung **.tagx** benutzt. TagDateien können wiederum aus Fragmenten von tag-Dateien zusammengesetzt sein. Solche Fragmente haben die Endung **.tagf**

12.5.2 Tag Dateien in der Web Application Structure

Tag Dateien können entweder als jar-File gepackt, oder ungepackt abgelegt werden.

TagFiles ungepackt ablegen

Es wird innerhalb des WEB-INF Ordners ein Ordner tags angelegt. Die tag-Dateien werden einfach ungepackt in dieses Verzeichnis abgelegt.

```
/WEB-INF/tags/
/WEB-INF/tags/a.tag
/WEB-INF/tags/b.tag
/WEB-INF/special/
/WEB-INF/special/c.tag
```

Werden die TagFiles ungepackt verwendet, muss in der Taglib Direktive der JSP statt des Attributs *uri* das Attribut *tagdir* verwenden.

```
<%@ taglib prefix="myPrefix"  tagdir="/WEB-INF/tags/special/" %>
```

Man muss keinen TagFileDescriptor (tld Datei) anlegen, da dies der Container selber macht.

Tag Dateien gepackt ablegen

Wenn die Tag Dateien als jar-Files gespeichert werden, sollten diese innerhalb des Verzeichnisses */WEB-INF/LIB* abgelegt werden. Das packen der Tag Dateien erleichtert das Deployment, erfordert aber einen zusätzlichen höheren Aufwand

- Die Tag-Dateien müssen in einem Verzeichnis **META-INF/tags** zusammengestellt werden.
- Es muss auch ein Tag File Descriptor für die TagDateien geschrieben werden. Dieser Tag File Descriptor wird mit den Tag Dateien gepackt.

Beispiel:

Vor dem Packen werden die oben genannten Tags folgendermassen abgelegt

```
helloapp
       |
       |  ...
       |
       |_ META-INF
       |   |
       |   |_ myTags.tld
       |   |
       |   |_ tags
       |        |
       |        |_ a.tag
       |        |_ b.tag
       |        |_ special
       |             |_ c.tag
       |
       |
       |_ WEB-INF
           |
           ...
```

Die Dateien im META-INF Verzeichnis sollten vor dem Deployment gepackt werden.

```
jar -cvf myTags.jar META-INF
```

Die erhaltene Datei myTags.jar kann bei der Ausführung in das Verzeichnis **WEB-INF/TAG**

Es fehlt noch den Aufbau der tagLib Descriptor Datei:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<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
     http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"

   <tlib-version>1.0</tlib-version>
   <short-name>myDefaultPrefixName</short-name>
   <uri>mytaglib</uri>

   <tag-file>
     <name>a</name>
     <path>/META-INF/tags/a.tag</path>
   </tag-file>

   <tag-file>
     <name>b</name>
     <path>/META-INF/tags/b.tag</path>
   </tag-file>

   <tag-file>
     <name>c</name>
     <path>/META-INF/tags/special/c.tag</path>
   </tag-file>
```

12.5.3 Aufbau der Tag Dateien

12.5.3.1 Zugriff auf Body und Attributübergabe

Mit der *tag* Direktive und ihrem Attribut *body-content* wird die Art des Bodys des Tags festgelegt.

```
<%@ tag body-content="empty" %>
```

Folgende Werte kann das Attribut body-content annehmen

empty	Der Tag darf kein Body haben, ansonsten gibt es einen Syntaxfehler
scriptless	Dies ist der Standard. Ein Body ist erlaubt, darf aber keine Skripts enthalten
tagdependent	Hier werden bei Auswertung des Bodys die EL Ausdrücke oder die

	Aktionen nicht ausgewertet, sondern als reinen Text gewertet

Innerhalb eines Tags kann man mit der Aktion **<jsp:doBody />** veranlassen dass der Body des Tags von der aufrufenden Seite ausgewertet wird.

Hier ein Beispiel greet.tag:

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

Hallo
<jsp:doBody var="userId"/%>
```

So kann greet in einer Jsp aufgerufen werden:

```
<%@ page contentType="text/html" %>
<%@ taglib="my" tagdir=/WEB-INF/tags/myTags" %>

<my:greet >
Klaus
</my:greet >
```

Die Aktion <jsp:doBody /> ohne Attribut gibt den Body der aufrufenden Seite aus. Ansonsten wird der Body der aufrufenden Seite in die Variable var gespeichert. Die Ausgabe erfolgt erst bei expliziter Auswertung der Variable var. Ist der auszulesender Body gross, so kann man auch einen varReader benutzen.

Die Aktion <jsp:doBody> hat folgende optionale Attribute

Attribut	**Beschreibung**
scope	Der Scope der Variable. Standard ist page
var	Name der Variable in der die Auswertung des Bodys geschrieben wird
varReader	Ein Reader Objekt mit der man auf die Auswertung des Bodys zugreifen kann

Mit der Direktive *attribute* werden von der aufrufenden Seite Attribute mit übergeben. Die Direktive *attribute* hat selbst folgende Attribute

name	**Eindeutiger Name für das Attribut**
required	false falls Attribut optional ist
fragment	true falls der Wert des Attributs nicht ausgewertet wird. So kann man auch JSP Elemente übergeben
rtexprvalue	Kann der Wert des Attributs dynamisch berechnet werden. Falls true ist z.B ein EL Ausdruck erlaubt
type	Laufzeittyp des Attributwerts. Standard ist String
description	Beschreibung des Attributs

Beispiel:

```
<%@ tag body-content="scriptless" %>
<%@ attribute name="sex" required=true%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test= ${sex == "m"}
```

```
  Hallo  Herr <jsp:doBody /%>
</c:if>

<c:if test=${sex == "f"}
  Hallo Frau <jsp:doBody /%>
</c:if>
```

Eine weitere Möglichkeit der Ausgabe ist die Aktion <jsp:invoke>. Diese Aktion wertet ein Fragment aus

Beispiel:

```
<%@ attribute name="pattern" fragment="true" %>

<%-- Auswertung und Ausgabe des Fragments --%>
<jsp:invoke fragment="pattern" />

<%-- Auswertung in eine Variable speichern --%>
<jsp:invoke fragment="pattern" var="result" />
```

12.5.3.2 Outputparamter

Die Direktive *variable*, die nur innerhalb Tag Dateien erlaubt sind, ermöglicht dass die aufrufende Seite auf Variable zugreifen können, die innerhalb der Tag Dateien gesetzt werden.

Eine Direktive *variable* hat folgenden Attribute:

name-given	Definiert eine Variable die von der aufrufenden Seite verwendet werden kann
name-from-attribute	Nur im Zusammenhang mit dem Attribut alias möglich. Es ist der Name eines Attributs das die aufrufende Seite verwendet, um den in der Tag-Datei gesetzten Wert auszulesen. In der Tag-Datei selber wird der gesetzte Wert an die in <alias> gebundene Variable weitergegeben. Das Attribut *alias* In diesem Attribut wird d
alias	Dies ist der Name der Variable innerhalb des Tags dessen Wert der aufrufende Seite übergeben wird. Dieser Wert wird in die Variable kopiert, die im Attribut *name-from-attibute* spezifiziert ist.
variable-class	Klassenname der Variable. Vorgabe String
declare	gibt an ob die Variable deklariert ist oder nicht. Vorgabe true
scope	Eine der Wert AT_END, AT_BEGIN oder NESTED, Vorgabe NESTED
description	Beschreibung

Die Attribute *variable-class*,*declare*,*scope* und *description* sind optional. Ansonsten gibt es 2 Formen der Verwendung der Direktive *variable*.

```
<%@ variable name-given="x" %>  oder
<%@ variable name-from-attribute="einAttributname" alias="x" %>
```

Das Attribut *sope* hat nichts mit den Sopes wie page,application usw. zu tun. Es difiniert die Sichtbarkeit der auszulesenden Variablen in der aufrufenden Seite. *scope* kann folgende Werte annehmen:

AT_BEGIN	Die Variable ist gleich nach dem StartTag sichtbar

AT_END	Die Variable ist erst nach Abarbeitung des Tags sichtbar
NESTED	Die Variable ist im TagBody sichtbar. Nach dem Tag nicht mehr

12.5.3.3 Erlaubte Direktiven innerhalb einer Tag Datei

Die meisten direktiven einer JSP Seite, sind auch in einer Tag Datei nutzbar. Nicht möglich in Tags ist die *page Direktive*, als Ersatz gibt es dafür die *tag-Direktive*

Direkive	**Bedeutung**
taglib	wie bei JSP
include	wie bei JSP, die includierten Dateien muessen der Syntax der Tag-Dateien entsprechen
tag	Ersatz für die page-Direktive bei JSP
attribute	Nur für Tags, handelt die Attributübergabe eines Tags
variable	Nur für Tags, dient dazu Werte von innen (Bearbeitung des Tags) nach aussen (Aufrufende JSP Seite) zu geben.

12.5.3.4 Implizite Variablen in einer Tag Datei

Ein Tag kann auf folgende impliziten Variablen zugreifen:

Object	**Type**
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
jspContext	javax.servlet.jsp.JspContext

Beachte das es innerhalb von Tags kein Zugriff auf eine pageContext Variable gibt, sondern auf die allgemeinere jspContext Variable. Es gibt auch kein Zugriff auf das page und das exception Objekt.

13 J2EE Patterns

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.

13.1 Front Controller

Wenn eine Anfrage direkt an die JSP's bzw. Servlets geht, kann es zu folgenden Problemen kommen.

- Jedes einzelne Servlet bzw. JSP braucht eigene Routinen z.B zur Authentifizieren
- Oft herscht eine Bildschirmablauf Logik, z.B. darf kann man nur eine Bestellung absenden, wenn etwas im Einkauskorb ist.Diese Ablauflogik ist auf die entsprechenden Servlets bzw JSP verteilt, da diese die nachfolgende kennen.

Die Idee des FrontControllers die Anfragen an einen Controller zu senden. Dieser Analysiert die Anfrage, und leitet diese dann ggf. an die Servlets bzw JSP die die Bildschirmoberfläche repräsentieren weiter.

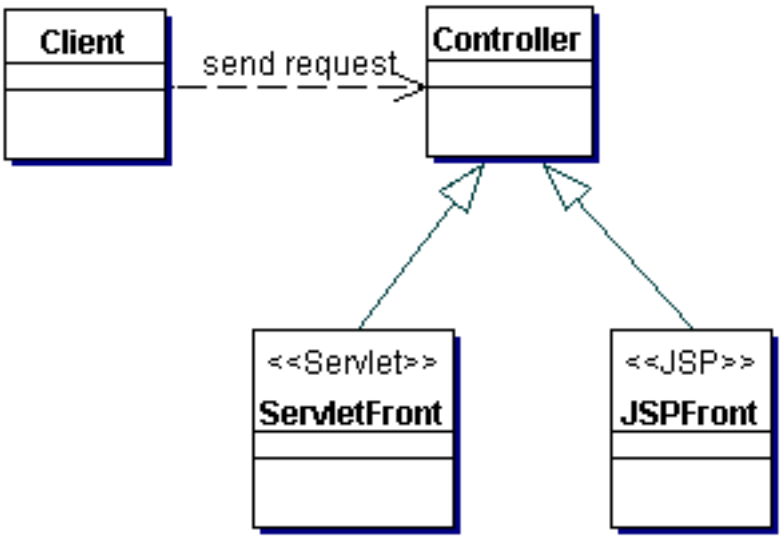


Abb. 13.1: Front Controller

Die Bildschirm-Ablauflogik kann nun zentral im FrontController behandelt werden. Der FrontController fungiert meistens als Dispatcher. Der Dispatcher leitet die Anfrage an die View (z.B: JSP) weiter. Helper Klassen (meistens als Java Bean) helfen dem Controller bzw. View ihre Arbeit zu erledigen.

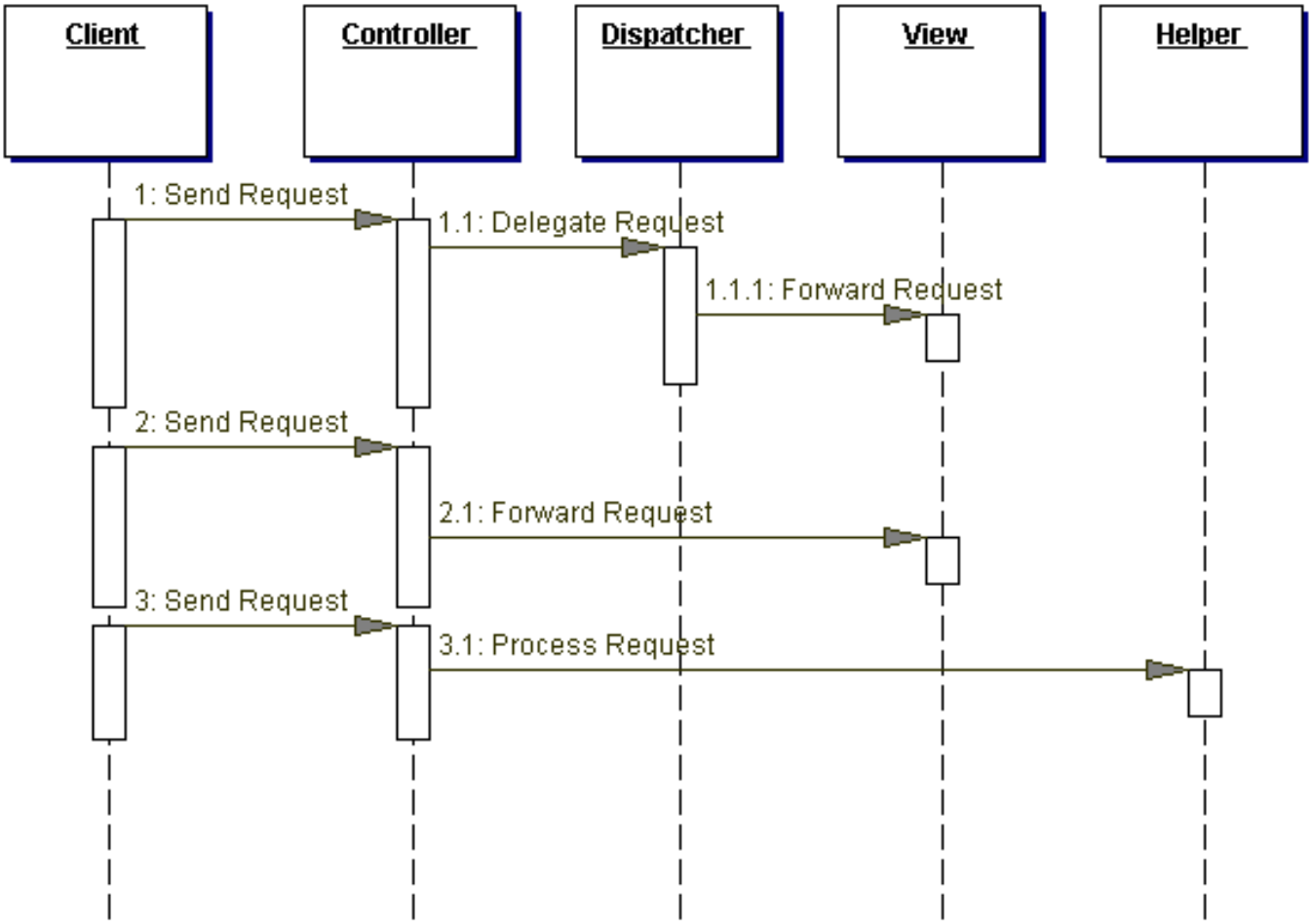


Abb. 13.2: Sequenzdiagramme Front Controller

Vorteile:

- Zentrale Kontrolle
- Verbessertes Sicherheitsmamanagement Durch den Controller hat man einen zentralen Einstiegspunkt. Dies erleichtert Logging Informationen und andere Sicherheitsmassnahmen.
- Erhöht Wiederverwendung Der Controller erleichtert es eine Applikation in mehreren Einheiten aufzuteilen. Funktionalität die von allen Servlets bzw JSP benötigt wird, kann zentral im Controller verarbeitet werden.

13.2 Transfer Object

Das primäre Ziel des Transfer Objects ist es die Netzwerkbelastung zu reduzieren. Statt einzelne Attribute eines Business Objects auszulesen, wird (auch wenn zunächst nur ein einzelnes Attribut gebraucht wird) werden i.a. alle Attribute des Business Objects gelesen. Diese Attribute werden aus dem Business Objekt in ein ValueObject kopiert. Dieses ValueObject wird dann versendet.

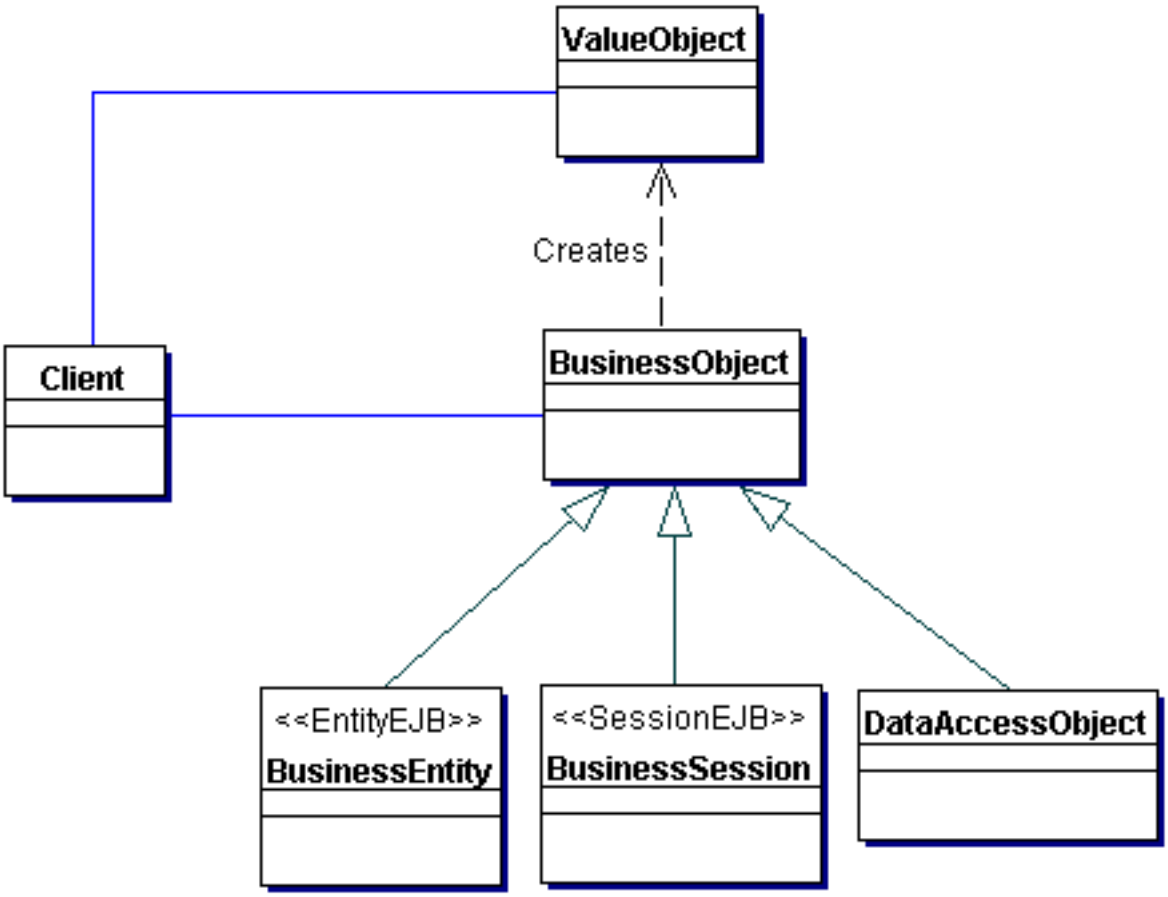


Abb. 13.3: Klassen Transfer Object

Verwende Transfer Object falls:

- Der Zugriff auf das Enterprise Bean über das Netz erfolgt. Die Zugriffe auf das Bean kann zu Netzwerk Overhead führen.
- Die Zahl der lesenden Zugriffe ist deutlich grösser als die schreibenden Zugriffe.
- Der Client benötigt meist mehrere Attribute eines Enterbrise Bean.

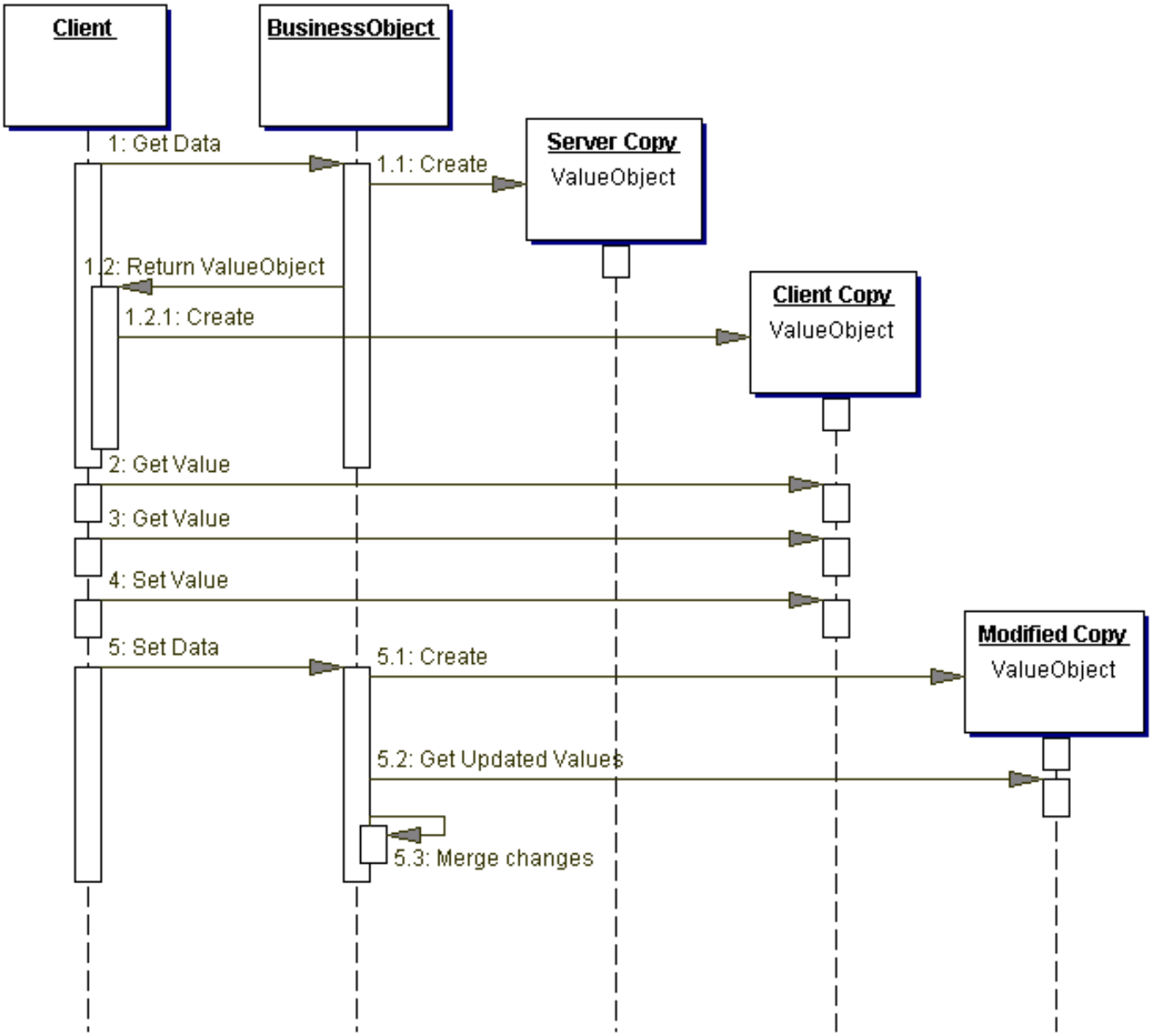


Abb. 13.4: SequenzDiagramm Klassen Transfer Object

Das obige Sequenzdiagramm zeigt einen möglichen Ablauf. Der Client holt aus dem BusinnessObject die Attribute, liest diese über ein Value Objekt modifiziert einige und sendet das Value Objekt wieder an den Server, bzw. aktualisiert das Value Objekt.

Beachte:

- Der Client kommuniziert mit dem BusinessObject mit dem Befehlen getData() zum Auslesen bzw setData() zum Senden der Attribute.
- Zum Übertragen der Attribute durch das Netz werden ValueObjekte genommen. Diese ValueObjekte müssen serialisiert werden können.
- Die Value Objekte sind sehr klein, sie werden i.a. wie bei ValueObjekten als ReadOnly Objekte realisiert. (im obigen Sequenzdiagramm wird dies mit Schritt 4 verletzt).

Vor- und Nachteile:

- Vereinfacht die Schnittstellen - Die Schnittstellen von den Clienten zu den Businness

Objekten sind identisch (setData bzw getData)
- Überträgt mehr Daten in weniger Zugriffen
- Reduziert die Netwerkbelastung
- Erhöht die Gefahr veraltete Objekte zu haben - Da man seltener auf den Server zugreift, ist die Gefahr dass ein anderer Client die Attribute auf dem Server geändert hat, grösser.
- Synchronisation und Versionskontrolle wird schwerer

13.3 Business Delegate

Business Delegate gehört zwar offiziel zu der Geschäftslogikschicht, lebt aber in der Präsentationsschicht. Business Delegate vereinfacht die Schnittstelle zu der Geschäftslogik, und entkoppelt Präsentationsschicht und Geschäftslogikschicht.

Das BusinessDelegate Objekt ist eine Art Fassaden Objekt für die Geschäftslogikschicht. Diese Fassadenobjekt ist allerdings auf der Clientenseite.

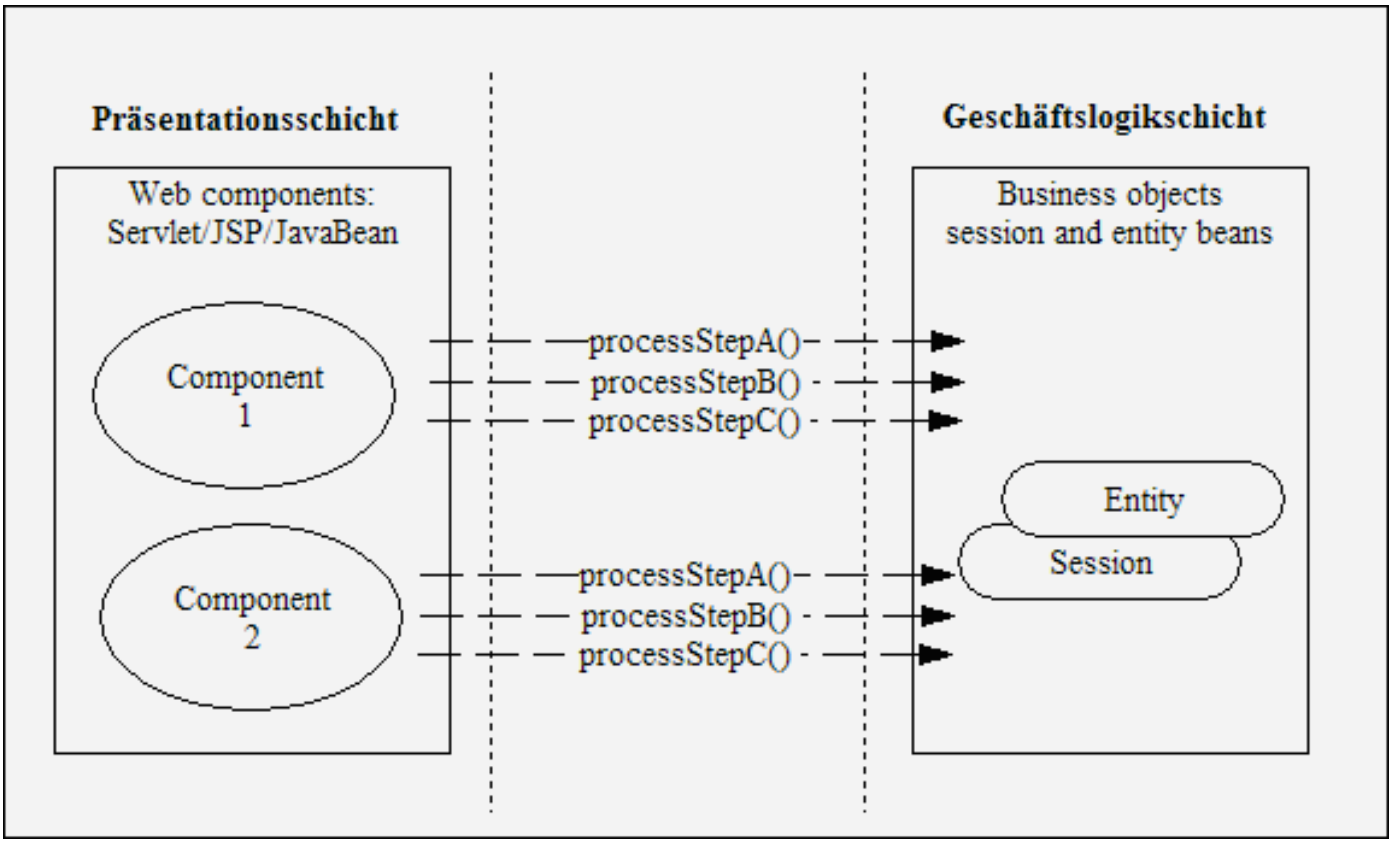


Abb. 13.5: Kommunikation ohne Business Delegate

Falls sich die API in der Geschäftslogik ändert, muss man in der obigen Abbildung Änderungen an Component1 und Component2 durchführen.

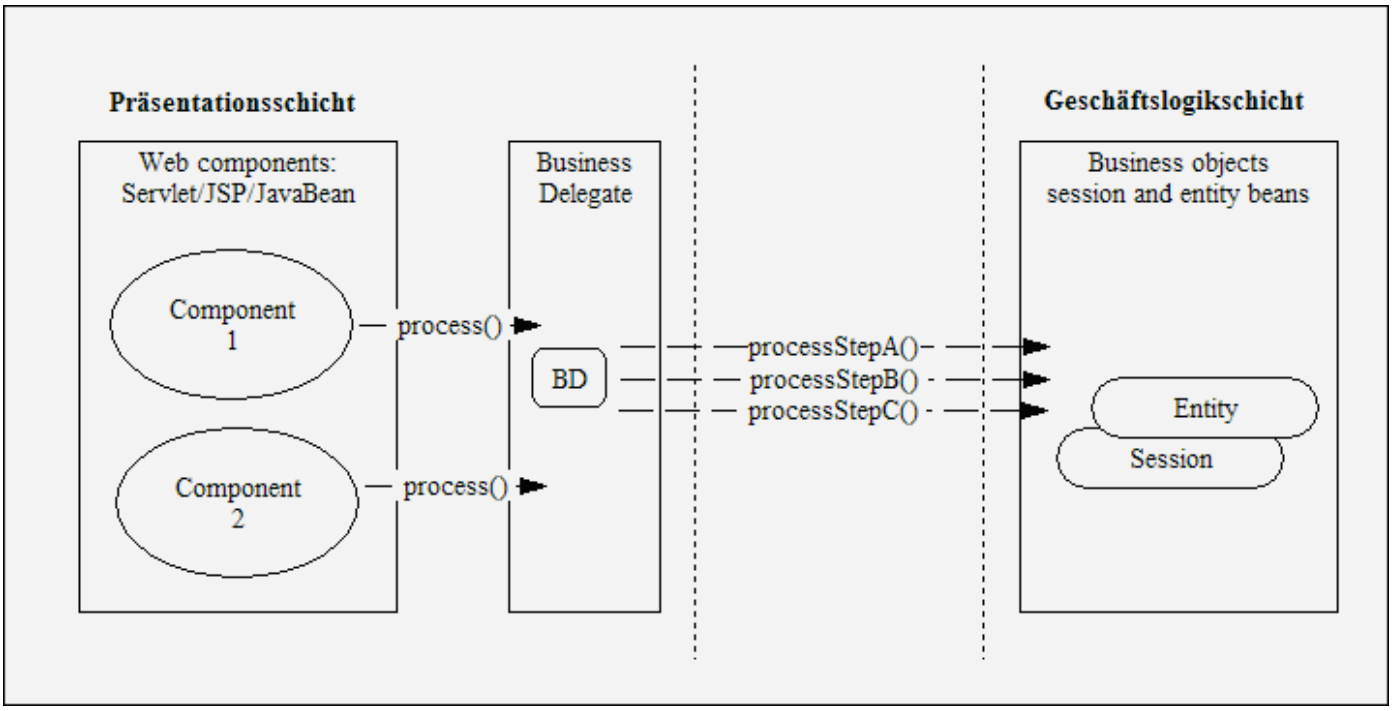


Abb. 13.6: Kommunikation mit Business Delegate

Mit einem Business Delegates Objekt muss bei Änderungen in der API der Geschäftslogik, Änderungen nur in im Business Delegates Objekt machen.

Verwende Business Delegate falls

- Die Objekte der Präsentationsschicht haben primär zwei Aufgaben
  - Die Benutzungsoberfläche zu managen
  - Zugriff auf die Geschäftslogik
- Der Code der die Benutzungsoberfläche repräsentiert nicht abhängig von der Geschäftslogik sein soll.
- Mehrere Komponenten der Präsentationsschicht dieselbe Sequenz von Aufrufen an die Geschäftslogikschicht haben.
- Es wird erwartet dass sich die API der Geschäftslogik ändert.

Vorteile:

- Entkoppelt Präsentationsschicht und Geschäftslogikschicht
- Kann als Proxy für den Client verwendet werden, und reduziert damit die Netzwerkbelastung
- Cacht Ergebnisse und Verweise der Geschäftslogik
- Vereinfacht als Fassaden Objekt die Schnittstelle zur Geschäftslogikschicht
- Kapselt den Zugriff auf die Geschäftslogikschicht.

13.4 Model View Controller

Trennt die Verantwortlichkeit beim Aufbau der Benutzungsoberflächen in die 3 unterschiedliche Rollen, um verschiedene Darstellungen derselben Information zu ermöglichen.

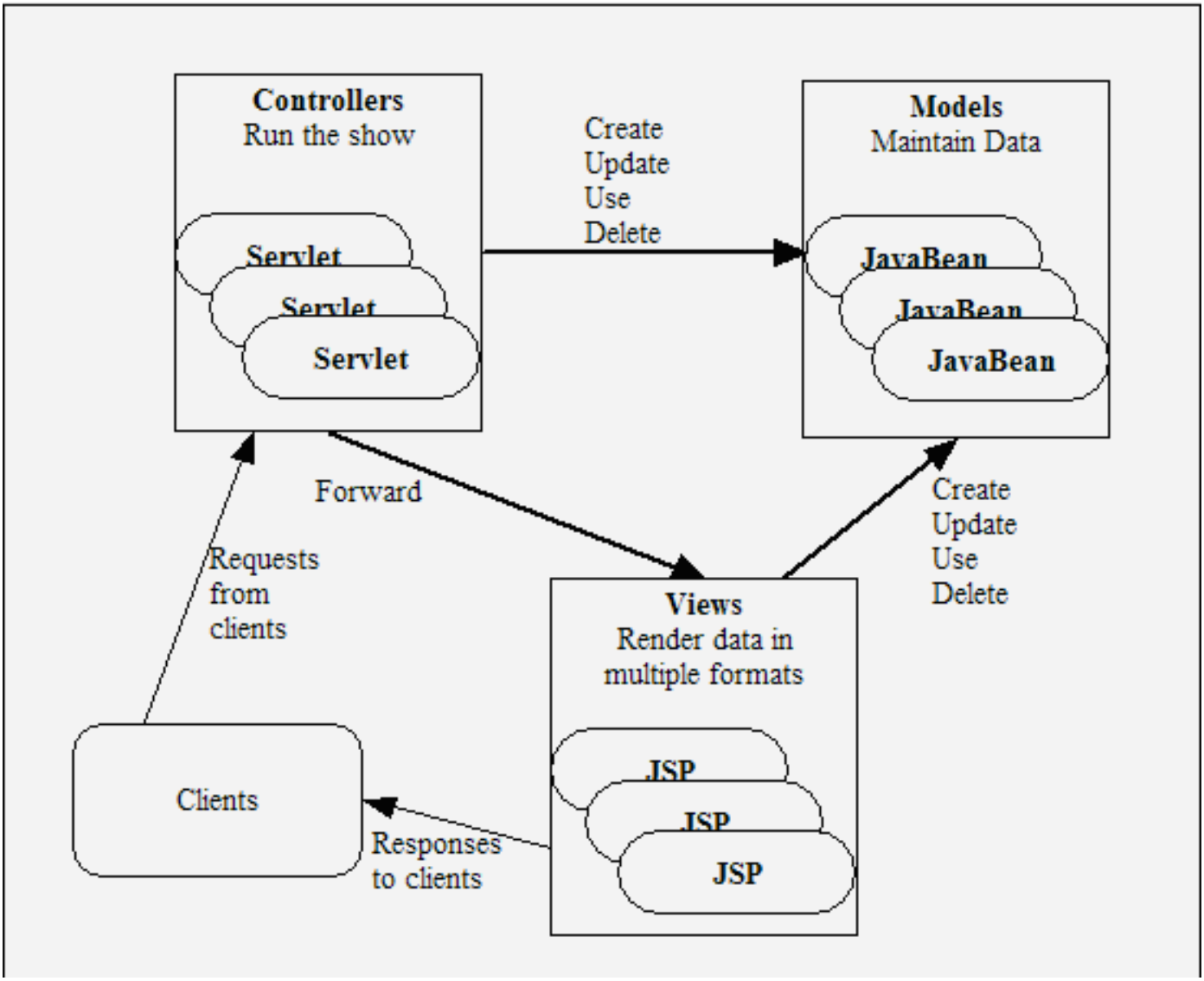


Abb. 13.7: Model View Controller Pattern

Verwende das MVC Pattern falls

- Es 3 verschiedene Arten von Aufgaben gibt
  - Die Interaktion des Benutzers mit dem System zu verwalten
  - Die darzustellende Information zu verwalten.
  - Diese Information soll unterschiedlich dargestellt werden
- Eine Komponente die all diese Aufgaben erfüllt, kann in 3 kleineren voneinander unabhängigen Komponenten aufgespaltet werden.

Die Komponenten

Das **Model** ist für die darzustellenden Informationen zuständig. Es persistiert diese Information. Das Model ist unabhängig von den Controllern und den Views. Das Model informiert die Views falls sich die Daten geändert haben. Da das Model seine View i.a. nicht kennt, geschieht dies durch das Beobachter Pattern. Wichtig ist dass die View sehr wohl sein Model kennt, das Model aber nicht die Views.Dies ermöglicht dass mehrere Views auf dasselbe Model zugreifen. Es ist aber nicht möglich, dass eine einzige View mehrere Models hat.

Die **View** ist für die Darstellung der Informationen zuständig.

Der **Controller** nimmt die Eingabe des Anwenders entgegen, bearbeitet das Modell und sorgt für die Aktualisierung der Anzeige. Controller und View arbeiten sehr eng zusammen. I.a. gibt es zu einer View genau einen Controller. Es kommt in der Praxis selten vor, daß nur der Controller ausgetauscht wird. Eine sinnvolle Anwendung wäre z.B. von einem editierbaren in ein nicht editierbaren Modus zu wechseln. In der Praxis sind meist Controller und View nicht getrennt und bilden eine Einheit. Diese Trennung zwischen View und Controller ist auch nicht so wichtig, wie die Trennung des Models von der View.

Vorsicht Missverständnis

Was die Rolle des Controllers und View betrifft gibt es einige Missverständnisse in der Literatur. Manche sehen in dem Controller ein Bindeglied zwischen Model und View. Die eigentliche Verarbeitung der Interaktion wird in dieser Literatur der View zugesprochen. Die Aufgabe des Controllers wird dann fälschlicherweise mit der Instantiierung des Models, oder Zuordnung des Models zu einer View zu erstellen. Der grosse Unterschied zum klassischen MVC Pattern liegt dann darin, dass es einen einzigen Controller zu mehreren Views gibt. Im klassischen MVC Pattern hat jede View seinen eigenen Controller. Der Controller der fälschlicherweise in der Literatur unter dem MVC Pattern angegeben wird, ist aber eine andere Art von Controller.

Vorteile des MVC Pattern

- Trennt die Darstellung von Information gegenüber der Verarbeitung
- Erleichter es veschieden Views einzusetzen, oder einfach die View auszutauschen.

13.5 Service Locator

Besorgt und cached alle J2EE-Ressourcen, die eine Komponente für die Abbildung der Geschäftslogik benötigt Typischerweise wird ein Service Locator für das Auffinden von Homes, LocalHomes oder DATA Sources verwendet.

Zum Auffinden von J2EE-Ressourcen wird meistens JNDI (Java Naming und Directory Interface) verwendet. Durch Verwendung des Service Locators müssen sich die Clients nicht mit JNDI herumschlagen.

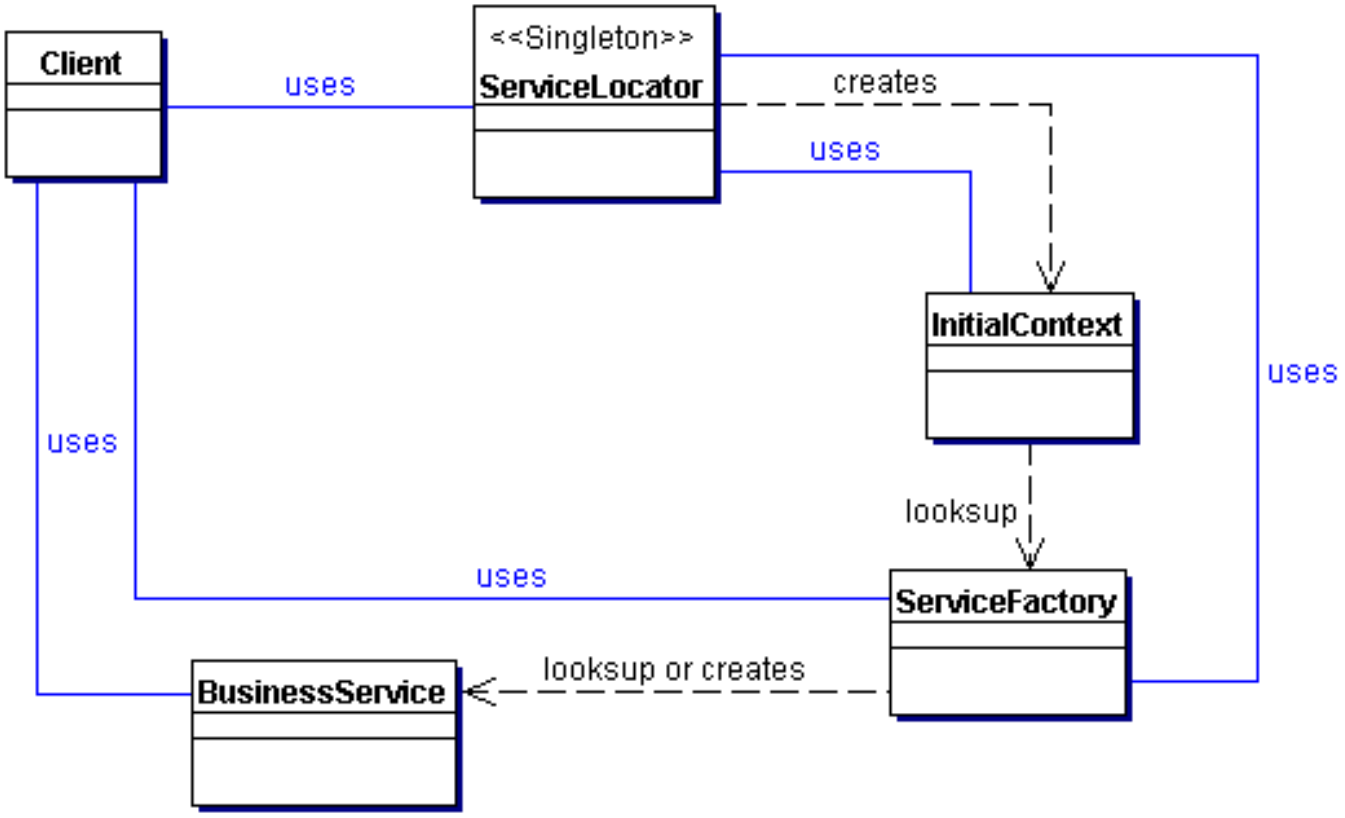


Abb. 13.8: Service Locator

Die Komponenten:

Ein **Client** braucht zur Bearbeitung seiner Aufgabe ein BusinessService Objekt. Er verwendet den **Service Locator** um dieses Objekt zu suchen und zu bekommen.

Der **ServiceLocator** übernimmt für den Clienten die Suche nach dem richtigen BusinessService Objekt.

Das **InitialContext** ist ein Einstiegspunkt für die Suche in einem Directory. Ein Directory ist baumartig strukturiert. Das InitialContext spezifiziert den Startknoten. Es wird dann in dem Directory von dem Startknoten aus gesucht, bis das BusinessService Objekt gefunden wird. Er erzeugt einen InitialContext um die Suche zu starten.

Das **ServiceFactory** wird benötig, da das BusinessService Objekt einen Lebenszyklus wie ein Servlet hat. D.h. ggf muss das **ServiceFactory** das BusinessService Objekt erst einladen, und instantiieren bevor es verwendet wird.

Das **BusinessService** Objekt bietet dem Clienten hoffentlich genau den Dienst an, den der Client benötigt.

Vorteile

- Versteckt Komplexität - Die Verantwortlichkeit eine JNDI Umgebung korrekt aufzubauen, wird dem Service Locator überlassen. Der Client muss sich nicht umd programmtechnische Details von JNDI kümmern.
- Bietet einheitlicher Zugriff auf Business Service Objekte - Der Service Locator bietet ein einfaches und präzises Interface an, das alle Clienten nutzen können. Es garantiert dass

alle Clienten der Applikation auf dieselbe Art und Weise Business Objekte finden bzw. erzeugen.
- Erleichtert das Hinzufügen neuer Business Service Objekte - Man kann neue EJBHome Objekte hinzufügen, ohne dass der Code bei den Clienten geändert werden muss.
- Reduziert Netzwerk Belastung - Da alle Anfragen der Clienten an JNDI durch den Service Locator gehen, können diese Anfragen gebündelt werden.
- Erhöhung der Performance - Das Cachen von z.B: des Initialen Contextes, Referenzen auf Objekte, verhindert das gleiche Aktionen doppelt ausgeführt wird.

13.6 Interception Filter

Filter haben Zugriff auf den Request und Response einer Webkomponente und ist somit in der Lage, ihre Funktionalität dynamisch zu erweitern. Mit Filter kann man sich vor und nach der eigentlichen Verarbeitung in einer Webkomponente einklinken.

Der Interception Filter erlauben es verschiedene Filter miteinander zu kombinieren. Filter hinzuzufügen oder zu entfernen ohne in den Code eingreifen zu müssen.

Filter wurden im Kapitel über das Web Container Model ausführlich behandelt.

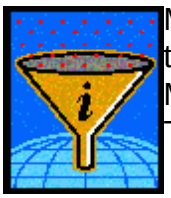
Vorteile:

- Zentrale Kontrolle für eine bestimmte Erweiterung - Ein Filter kapselt zentral eine Erweiterung zB. Authentifizierung oder Logging, diese Erweiterung kann man dann für beliebige Webkomponenten benutzen. Filterketten erlauben diese Erweiterungen beliebig zu kombinieren, da die einzelnen Filter und die Webkomponenten sehr gering miteinander gekoppelt sind.
- Erhöht Wiederverwendbarkeit - Da die Filter sehr flexibel kombiniert werden können, können derselbe Filter auch in mehreren Kontexten verwendet werden.
- Deklerative und Flexible Konfiguration - Die Konfiguration der Filter erfolgt in der Deployment Descriptor Datei web.xml. D.h. man kann Filter kombinieren ohne den Code ändern zu müssen.

Nachteile:

- Da Filter sehr loose miteinander gekoppelt sind, ist es schwierig Informationen zwischen den Filtern zu teilen.

13.7 Zusammenfassung der Vorteile für die Patterns



Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.

Interception Filter

1. Pre- und Postprocessing
2. Zentrale Kontrolle für eine bestimmte Erweiterung
3. Erhöht Wiederverwendbarkeit
4. Deklerative und Flexible Konfiguration

Model View Controller

1. Flexibler Design
2. Trennung Interaktion, Präsentation, und Geschäftsobjekte
3. Möglichkeit von mehreren Präsentationen

Front Controller

1. Zentrale Kontrolle über Bildschirmablauflogik
2. Verbessertes Sicherheitsmamanagement
3. Erhöht Wiederverwendung

Service Locator

1. Kapselt Komplexität zum Auffinden von Diensten
2. Bietet Clienten einfaches und einheitliches Interface zum Zugriff auf Dienste
3. Reduziert Netzwerk Belastung duch Cachen der Ergebnisses oder bündeln der Suchanfragen.
4. Erhöhung der Performance

Business Delegate

1. Geringere Kopplung von Präsentationsschicht und Geschäftslogikschicht
2. Entkoppelt Clienten von API der Business Objekte
3. Zugriff auf Business Objekte ist gekapselt
4. Client seitige Fassade

Transfer Object

1. Verringert Netzwerkbelastung
2. Reduziert Kommunikation über das Netz
3. Verbessert Performance bei Anfragen