

## Einführung in die Konzepte von JavaServer Faces (JSF)

von Muhammet Öztürk und Marco Michel

# GUI fürs Web

Derzeit existieren über dreißig vergleichbare Web-Frameworks auf dem Markt, allerdings existiert bis heute kein gemeinsamer Standard. Diese aktuelle Vielfalt unterschiedlicher Frameworks macht es den Toolherstellern nicht leicht, Support für das eine oder andere dieser Frameworks zu implementieren. Mit den JavaServer Faces hat Sun ein Framework geschaffen, das gute Aussichten hat, diesem Mangel Abhilfe zu schaffen.



In diesem ersten Teil unseres Schwerpunkts erläutern wir ausführlich die Konzepte, die hinter JSF stehen. Im zweiten Teil folgt ein Beispiel, das die im ersten Teil dargestellten Eigenschaften des Frameworks noch einmal anhand einer beispielhaften Web-Anwendung vorstellt.

### JavaServer Faces

Die JavaServer Faces-Technologie, deren erste Spezifikation Sun im September 2002 unter dem JSF 1.2 vorstellte, basiert natürlich auf dem bekannten Model View Controller (MVC)-Modell und liegt derzeit zwar noch nicht als Finalversion, jedoch als relativ ausgereifte Spezifikation vor. Wir gehen davon aus, dass bis zur endgültigen Verabschiedung der Spezifikation nur noch ein paar „Schönheitsfehler“ korrigiert werden.

Obwohl sich die J2EE (zu Recht) als Vorbild für die .NET-Plattform sieht, ist

bei den JavaServer Faces nicht zu verkennen, dass ihr Konzept in wesentlichen Zügen dem WebForms-Framework aus .NET entliehen ist. Zusätzlich zu diesen Ähnlichkeiten wird man viele Konzepte, wie z.B. das Rendering und das Eventhandling, die man bereits von Java Swing kennt, wiederfinden.

JSF verfügen über wohldefinierte Phasen, die den Lebenszyklus der JSF bilden, um die Präsentation einer Web-Anwendung zu verwalten. Im Folgenden werden wir den Lebenszyklus bzw. diese Phasen ausführlich unter die Lupe nehmen.

### Aufbau von JSF

JSF bestehen aus den folgenden zwei Hauptkomponenten:

- Den JSF APIs und einer Referenzimplementierung – in Zukunft werden wahrscheinlich einige Toolhersteller eigene

Implementierungen zur Verfügung stellen und dabei die so genannten serverseitigen Komponenten *UIComponent*, *Renderer*, *Converter*, *Validator*, *Listener* und *JSF-Runtime* definieren bzw. implementieren.

- Den JSF Tag Libraries, die diese Komponenten in einer *JavaServer Page* abbilden.

JSF bieten eine klare Trennung zwischen dem Verhalten bzw. dem Zustand eines *UIComponent*-Objekts und dessen Darstellung. Während eine *UIComponent* mittels JSF Tag Libraries in einer *JSP* dargestellt wird, wird ihr Verhalten bzw. ihr Zustand auf der Server-Seite durch ein *UIComponent*-Objekt mit jeweiligen *Renderer*-, *Converter*-, *Validator*- und *Listener*-Objekten repräsentiert.

JSF Tag Libraries dienen als Brücke zwischen *JSPs* und Komponenten, was eine Wiederverwendbarkeit von Kompo-



Listing 1

Quellcode der Seite *anmeldung.jsp*

```
<%@ page contentType="text/html; charset=
    iso-8859-1" language="java" %>
<%@ taglib uri="/WEB-INF/fmt-1_0.tld" prefix="fmt" %>
<%@ taglib uri="/WEB-INF/html_basic.tld" prefix="h" %>
<%@ taglib uri="/WEB-INF/jsf_core.tld" prefix="f" %>
<%@ taglib uri="/WEB-INF/c-1_0.tld" prefix="c" %>

<fmt:setBundle basename="de.wwag.arch.alfa.
    presentation.level0.apps.jsfClient.Resources"
    scope="session" var="PR_Bundle" />
<f:use_faces>
<h:form id="anmeldung.jsp" formName="anmeldung" >
<title>
<h:output_text id="title" key="anmeldung.
    title" bundle="PR_Bundle" />
</title>
<table class="defaulttype" cellspacing=
    "0" cellpadding="0" border="0">
<tr>
<td>
<h:output_text id="label_bk" key="anmeldung.
    benutzerkennung"
    bundle="PR_Bundle" />
</td>
<td>
<h:input_text id="input_bk" valueRef="ablaufForm.
    benutzerkennung" />
</td>
</tr>
<tr>
<td>
<h:output_text id="output_kw" key="anmeldung.
    kennwort"
    bundle="PR_Bundle" />
</td>
<td>
<h:input_secret id="input_kw" redisplay="false"
    valueRef="ablaufForm.kennwort" />
</td>
</tr>
<tr>
<td>
<h:command_button id="command_an" key=
    "anmeldung.anmelden"
    bundle="PR_Bundle" commandName="anmelden" >
<f:action_listener type="de.wwag.arch.alfa.
    PRActionListener" />
</h:command_button>
</td>
</tr>
</table>
</h:form>
</f:use_faces>
```

Component-Objekt auszuführen. Man kann sich dazu das *UIComponent*-Objekt als ein Auto vorstellen, dessen Lebenszyklus als ein Fließband und die Phasen als bestimmte Bereiche dieses Fließbandes. In diesen verschiedenen Phasen werden die jeweiligen Methoden des *UIComponent*-Objekts aufgerufen, um seinen Zustand zu managen bzw. zu ändern. Der Entwickler einer Komponente hat immer die Möglichkeit, den Lebenszyklus abubrechen oder direkt zur Rendering-Phase zu springen.

Grundsätzlich findet man in einer JSF-Anwendung drei Szenarien, die unterschiedliche Phasen nach unterschiedlichen Reihenfolgen durchlaufen:

- Non-Faces Request generiert Faces Response,
- Faces Request generiert Non-Faces Response,
- Faces Request generiert Faces Response.

Um diese Szenarien bzw. Phasen besser beschreiben zu können, möchten wir sie anhand eines kleinen Beispiels kurz vorstellen. Abbildung 3 zeigt eine einfache HTML-Seite, die Benutzerkennung und -kennwort abfragt. Listing 1 zeigt den Quellcode der Seite *anmeldung.jsp*.

Bevor man eine JSF-Applikation auf einem Webserver deployt, müssen alle passenden Request-URLs in der Datei *web.xml* der Anwendung mit dem folgenden Eintrag an das Servlet *javax.faces.webapp.FacesServlet* weitergeleitet (gemappt) werden:

```
<!-- Faces Servlet Mapping -->
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Nehmen wir an, dass diese Anwendung auf dem gleichen Server läuft und mit der URL *http://localhost:8080/My-Applikation/jsp/anmeldung.jsp* als so genannter Non-Faces Request aufgerufen wird. JSF nennt den Request deshalb Non-Faces, weil er von einer „nicht-Render Response“-Phase gerenderten Seite stammt. Zunächst wird überprüft, ob für diesen Request mit der TreeID */jsp/anmeldung.*

*jsp* bereits ein Baum (Abb. 3) im aktuellen *FacesContext*-Objekt existiert. Da die Anwendung in unserem Beispiel zum ersten Mal gestartet wird, findet JSF hier kein Tree-Objekt.

Falls ein Tree-Objekt für diese Request-Identifikation im aktuellen *FacesContext*-Objekt bereits existiert, bedeutet das, dass hier ein „Faces Request“ vorliegt, da dieser Request von der „Render Response“-Phase einer gerenderten Seite gestellt wurde. Im Folgenden werden wir diese Szenarien in ihren einzelnen Phasen ausführlich beschreiben.

Die Phase „Reconstitute Component Tree“

Im Falle eines „Non-Faces Request“ wird zunächst aus der jeweiligen JSP ein *Tree*-Objekt mit den ggf. vorhandenen Kindobjekten (Abb. 3) neu erzeugt und im *FacesContext*-Objekt abgelegt, um beim nächsten Request wieder darauf zugreifen zu können. Folgender Codeabschnitt zeigt, wie man in einem Listener ein *Tree*-Objekt für die URL *anmelden.jsp* erzeugt und dieses im aktuellen *FacesContext* registriert.

```
//Gewünschte TreeID
String treeId = "anmelden.jsp"
TreeFactory tFactory = (TreeFactory)FactoryFinder.
    getFactory(FactoryFinder.TREE_FACTORY);
Tree tree = tFactory.setTree(facesContext, treeId);
```

Anschließend wird die Methode *renderResponse()* der Klasse *FacesContext* von JSF aufgerufen, um direkt die Phase „Render Response“ auszuführen. Es hat hier natürlich keinen Sinn, die weiteren Phasen zu durchlaufen, weil das *Tree*-Objekt ja neu erzeugt wurde. Das neu erzeugte *Tree*-Objekt wird im aktuellen *FacesContext*-Objekt registriert und zum Rendern in die „Render Response“-Phase gewechselt. In diesem Fall handelt es sich um ein „Non-Faces Request generiertes Faces Response“-Szenario. Wie Sie auch in Abbildung 5 sehen, ist bei diesem Szenario die Phase „Reconstitute Component Tree“ nicht involviert.

Wenn ein *Tree*-Objekt für die aktuelle Request-Identifikation existiert, der Request also von einer Seite geschickt wird, die bereits von einem JSF-Renderer bear-

beitet wurde, ändert sich auch die Reihenfolge der Ausführung der Phasen. Es handelt sich in diesem Fall um das Szenario „Faces Request generiert Faces Response“, das uns bei JSF am meisten interessieren wird. Dies wird auch als „Standard Request Processing Lifecycle“ bezeichnet, da solche Requests normalerweise alle Phasen von JSF durchlaufen sollen. In diesem Fall wird die Lifecycle Management-Methode *processReconstitutes()* von der zuvor erzeugten Root-*UIComponent* aufgerufen. Die Aufrufe aller Lifecycle Management-Methoden erfolgen nach dem Composite Design-Pattern, welches es dem Client – in diesem Fall *FacesServlet* – ermöglicht, sowohl das Root-Objekt als auch die Kinderobjekte einheitlich zu behandeln. Der Aufruf der Methode *processReconstitutes()* des Root führt dazu, dass die gleichen Methoden aller Kinderobjekte nacheinander aufgerufen werden. Während der Phase „Reconstitute Component Tree“ werden alle in einer JSP mittels JSF Tag Library eingegebenen Converter, Validators und Listener erzeugt und an das *UIComponent*-Objekt im *Tree*-Objekt angehängt. In unserem Beispiel besteht das *Tree*-Objekt aus sechs Kinderobjekten. Zusätzlich wurde ein Listener-Objekt von der Klasse *de.wvag.arch.alfa.PRActionListener* erzeugt und für das Kinderobjekt mit der ID *command\_button*

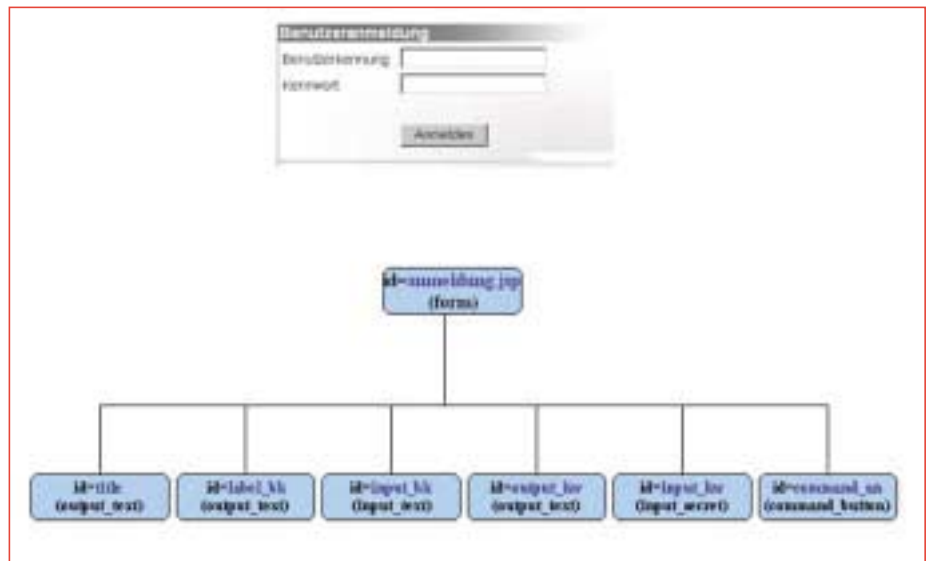


Abb. 3: Komponentenbaum der Seite *anmeldung.jsp*

registriert (Abb. 3 und Listing 1). Nach der Ausführung dieser Phase wird die nächste Phase „Apply Request Values“ ausgeführt.

### Die Phase „Apply Request Values“

Die „Apply Request Values“-Phase des Standard Request Processing Lifecycle ermöglicht es jeder Komponente im *Tree*-Objekt, ihren eigenen aktuellen Wert aus dem Request herauszuholen und ihn als lokalen Wert mit neuem Inhalt zu aktualisieren. In dieser Phase ruft JSF die Life-

cycle Management-Methode *processDecodes()* der Root-UI-Komponente auf (Abb. 5). Dieser Aufruf veranlasst, dass die gleiche Methode von den Kinderobjekten aufgerufen wird. Die Methode *processDecodes()* ruft intern die Component Spezialisierung-Methode *decode()* des jeweiligen Komponenten- bzw. Renderer-Objekts auf (Abb. 2). Eine *UIComponent* kann entweder sich selbst rendern, sodass man die *decode()*- bzw. *encodeXXX()*-Methode des Interfaces *UIComponent* überschreibt, oder man kann diesen Vorgang

## Anzeige

einem Renderer überlassen, der diese beiden Methoden implementiert. Diese letztere Variante wird von JSF empfohlen, da man damit die Darstellung einer Komponente von ihrem Verhalten bzw. Zustand trennt. Um herauszufinden, ob eine Komponente einen Renderer hat, wird die Methode `getRendererSelf()` der jeweiligen UI-Komponente aufgerufen. Wenn der Aufruf dieser Methode `true` zurückliefert, wird `decode()` von dieser UI-Komponente aufgerufen; ansonsten erfolgt ein Aufruf von `decode()` durch den Renderer der jeweiligen UI-Komponente. Der Renderer einer Komponente wird während der Erzeugung des *Tree*-Objekts durch den Aufruf der Komponentemethode `getRendererType()` ermittelt bzw. erzeugt und für die Komponente registriert.

Beim `decode()`-Aufruf durch den Renderer oder die Komponente überprüft JSF, ob diese Komponente Konverter hat. Wenn für diese Komponente Konverter existieren bzw. registriert sind, wird die Methode `getAsObject()` der jeweiligen Konverter aufgerufen, um den Wert aus dem Request zu extrahieren und in einen lokalen Variabelentyp zu konvertieren. JSF stellt den Entwicklern mehrere Standard-Konverter zur Verfügung und ermöglicht es auch, eigene zu schreiben. Die eigenen Konverter-Klassen müssen das Interface *Converter* implementieren. Der folgende Code-Abschnitt zeigt, wie ein selbstentwickelter `de.wvag.alfa.EmailConverter` an eine `input_text`-Komponente anhängt wird:

```
<h:input_text id="mxText" size="16" converter="
    "de.wvag.alfa.EmailConverter">
```

Das Interface *Converter* deklariert die beiden public Objekt-Methoden:

```
getAsObject(FacesContext context,
    UIComponent comp, String newValue)
getAsString(FacesContext context, UIComponent comp,
    String newValue)
```

Diese beiden Methoden definieren eigentlich zwei Sichten auf die Modelldaten. Der Aufruf der Methode `getAsObject()` des jeweiligen Konverters konvertiert den übergebenen Wert vom Request in eine lokale Variable der Komponente. Diese lo-

kale Variable der UI-Komponente wird in der Phase „Update Model Values“ ins Modell kopiert. Falls ein Fehler beim Aufruf der Methode `getAsObject()` auftritt, wird dieser durch den Aufruf `addMessage()` vom aktuellen *FacesContext*-Objekt gespeichert. Wenn JSF bei der Ausführung der Phase „Process Events“ einen Fehler im aktuellen *FacesContext*-Objekt entdeckt, ruft dieses die Methode `renderResponse()` auf, um die aktuelle Seite mit dem Fehler zu rendern (Abb. 5). In Listing 6 wird die Implementierung der `decode()`-Methode einer Renderer-Klasse gezeigt. Der Wert `index` wird aus dem aktuellen Request gelesen, konvertiert und in die lokale Variable der Komponente übernommen.

Wenn eine der `decode()`-Methoden vom Renderer bzw. der UI-Komponente im *Tree*-Objekt die Methode `responseComplete()` vom aktuellen *FacesContext*-Objekt aufruft, wird der Lifecycle-Prozess für den aktuellen Request abgebrochen. Wird die Methode `renderResponse()` vom aktuellen *FacesContext*-Objekt in der Methoden `decode()` vom Renderer bzw. der UI-Komponente im *Tree*-Objekt aufgerufen, wird zur Phase „Render Response“ gewechselt, um die aktuelle Seite zu rendern. Ansonsten wird die nächste Phase „Process Events“ ausgeführt.

### Die Phase „Process Events“

Bei der Ausführung der Phasen „Apply Request Values“, „Process Validations“ und „Update Model Values“ besteht die Möglichkeit, Events auszulösen. Um diese Events an den jeweiligen Listener weiterzuleiten, wird nach der Ausführung der eigentlichen Phase die Phase „Process Events“ des Standard Request Processing Lifecycle ausgeführt.

Während dieser Phase wird von JSF die Methode `getFacesEvents()` vom aktuellen *FacesContext*-Objekt aufgerufen, um herauszufinden, ob Events von der vorherigen Phase ausgelöst wurden. Wenn dieser Aufruf ein *Events*-Objekt zurückliefert, ruft JSF die Methode `getComponent()` aller zurückgelieferten Event-Objekte auf, um die Methode `broadcast()` der jeweiligen Komponente aufzurufen. `broadcast()` ruft alle registrierten Listener der jeweiligen UI-Komponente auf, um sie

zu benachrichtigen, dass ein Event ausgelöst wurde.

### Die Phase „Process Validations“

In dieser Phase wird von JSF die Lifecycle-Management-Methode `processValidators()` von der Root-UI-Komponente aufgerufen. Dieser Aufruf führt dazu, dass die Validatoren der jeweiligen UI-Komponente den Inhalt der lokalen Variablen überprüfen. Während der Erzeugung des *Tree*-Objekts in der Phase „Reconstitute Component Tree“ werden auch die Validatoren für jede UI-Komponente erzeugt und registriert. Sowohl JSF als auch der Standard-Konverter stellen Standard-Validatoren zur Verfügung und ermöglichen es, eigene Validatoren zu schreiben. Diese Validator-Klassen müssen vom Interface *Validator* abgeleitet werden. Dieses Interface besitzt die Methode `validate()`, in der man den lokalen Wert der Komponente nach in den JSP definierten Regeln überprüft. Falls ein Fehler in

#### Listing 2

##### Die Methode `updateModel` einer UI-Komponente

```
public void updateModel(FacesContext context)
    throws IOException {
    counter=0;
    if (context == null) {
        throw new NullPointerException();
    }
    //der lokaler Wert der UI-Komponente
    String index = this.getIndex();
    //die durch "valueRef" referenzierten Attribute des Modells
    //werden aktualisiert
    Util.getValueBinding(getValueRef()).
    setValue(FacesContext.getCurrentInstance().value);
    //der lokaler Wert der UI-Komponente wird auf null
    //gesetzt
    this.setIndex(null);
}
```

#### Listing 3

##### Die UI-Komponente `command_button` mit Standard-Listener

```
<td>
<h:command_button id="command_an" key=
    "anmeldung.anmelden"
    bundle="PR_Bundle" commandName="anmelden"
    actionRef="ablaufform.anmelden">
</h:command_button>
</td>
```

dieser Phase auftritt, wird er durch den Aufruf `addMessage()` des aktuellen `FacesContext`-Objekt gespeichert. Wenn JSF in der folgenden „Process Events“-Phase einen Fehler im aktuellen `FacesContext`-Objekt entdeckt, ruft dieses die Methode `renderResponse()` auf, um die aktuelle Seite mit der Fehlermeldung zu rendern.

Wird beim Aufruf von `validate()` ein Event ausgelöst, so werden alle Listener der jeweiligen Komponente in der nächste Phase „Process Events“ benachrichtigt.

Wenn eine der `validate()`-Methoden des Validators die Methode `responseComplete()` des aktuellen `FacesContext`-Objekts aufruft, wird der Lifecycle-Prozess für den aktuellen Request abgebrochen. Ansonsten wird die nächste Phase „Process Events“ ausgeführt.

### Die Phase „Update Model Values“

Nun wird die Lifecycle Management-Methode `processUpdates()` von der Root-UI-Komponente aufgerufen. `processUpdates()` ruft intern die Component Specialization-Methode `updateModel()` der jeweiligen UI-Komponente auf. Wenn diese Phase erreicht wird, bedeutet es, dass die lokale Werte aller UI-Komponenten in dem `Tree`-Objekt aktualisiert wurden. In dieser Phase werden die lokalen Werte aller Input-UI-Komponenten, die als `valueRef`-Attribute in der JSP vorbelegt sind, in die von `valueRef` referenzierten Attribute des Modells kopiert. Am Ende dieser Phase sind alle diese Attribute aktualisiert und die lokale Variablen der jeweiligen UI-Komponenten gelöscht.

In Listing 2 wird die lokale Variable einer UI-Komponente in die referenzierten Attribute des Modells kopiert. Falls ein Fehler in dieser Phase auftritt, wird er durch den Aufruf `addMessage()` des aktuellen `FacesContext`-Objekt gespeichert. Wenn JSF in der folgenden Phase „Process Events“ einen Fehler im aktuellen `FacesContext`-Objekt entdeckt, wird die Methode `renderResponse()` des aktuellen `FacesContext`-Objekts aufgerufen, um die aktuelle Seite mit der Fehlermeldung zu rendern (Abb. 5).

### Die Phase „Invoke Application“

In dieser Phase wird die Business Logik aufgerufen und das Modell mit den Rückgabe-

werten des Aufrufs aktualisiert. Während dieser Phase werden alle Action Events vom jeweiligen Listener behandelt. Diese Events werden von einem Button oder Hyperlink – den so genannten `UICommand`-Komponenten – ausgelöst. In Listing 1 ist zu sehen, wie ein `command_button` mit dem Listener `de.uwag.arch.alfa.PRActionListener` und dem Eventnamen `anmelden` registriert wird. Die Listener-

## Ein Standard-Listener verwaltet die Navigation

Klasse muss das Interface `javax.faces.event.ActionListener` implementieren, dessen Methode `processAction(ActionEvent event)` im Falle eines Events nach dem Command Pattern-Prinzip aufgerufen wird. Man kann den in JSP mittels JSF-Tags festgelegten Eventnamen in der Listener-Methode `processAction(Action-Event event)` durch den Aufruf der Methode `getActionCommand()` der Klasse `Action-Event` dahingehend abfragen, ob ein gewünschter Button geklickt wurde. Das Interface `ActionListener` hat noch die wichtige Methode `getPhaseId()`, welche die Phase identifiziert, nach deren Ausführung der entsprechende Listener aufgerufen werden muss. Liefert die Methode `getPhaseId()` eine `PhaseId.ANY_PHASE` zurück, wird dieser Listener in jeder „Process Events“-Phase aufgerufen, wenn ein Event ausgelöst wird (Abb. 5).

JSF stellt einen Standard-`ActionListener` zur Verfügung, der die Navigation der Seiten einer Anwendung automatisch verwaltet, sobald man in der JSP für die jeweilige UI-Komponente eine Methode für `actionRef` festlegt. Dieser Standard-Listener ruft dann zunächst die in der `actionRef` referenzierte Businesslogik-Methode auf. Diese so genannte Businesslogik-Methode muss ein `Action`-Objekt zurückliefern, dessen Methode `invoke` von JSF nach dem Command-Pattern aufgerufen wird. Mit `invoke` eines `Action`-Objekts wird die Business Logik aufgerufen, bzw. das Modell aktualisiert (Listing 13). Was wir in Listing 1 durch eine UI-Komponente `command_button` mit dem Listener aus Listing 4 erreicht haben, können wir auch mit Listing 3 und 13 erreichen.

Es ist zu empfehlen, die zweite Variante, den Standard-Listener zu verwenden, wenn man von einer aktuellen Seite zu einer anderen wechseln muss. Dagegen sollte man sich für eigene Listener entscheiden, wenn man die Daten der entsprechenden Seite aktualisiert, z.B. wenn die Businesslogik aufgerufen wird, ohne das aktuelle `Tree`-Objekt in `FacesContext` zu ändern. Natürlich haben Sie auch die Möglichkeit, in einem eigenen Listener nach Aufruf der Businesslogik dem `NavigationHandler` mit Hilfe eines `Action`-Objekts mitzuteilen, welche Seite anzuzeigen ist. Der `NavigationHandler` entscheidet dann je nach Rückgabewert des Aufrufes von `invoke()`, von welcher JSP er das anzuzeigende `Tree`-Objekt erzeugt und in wel-

## Anzeige

chem *FacesContext* dieses ablegt wird. Da die Seitennavigation von JSF ein wichtiger Teil dieses Frameworks ist, werden wir dieses Thema separat in dem Kapitel „JSF Navigation Model“ behandeln.

Wenn man in dieser Phase die Methode *responseComplete()* des aktuellen *FacesContext*-Objekt nicht aufruft, führt dieser Aufruf zum Abbrechen des Lifecycle-Prozesses für den aktuellen Request, und die JSF leitet die nächste Phase „Render Response“ ein.

### Die Phase „Render Response“

Während dieser Phase wird das *Tree*-Objekt mit den ganzen Kinder-UI-Komponenten, die in dem aktuellen *FacesContext* in der Phase „Invoke Application“ gespeichert wurden, gerendert. Je nachdem, ob eine Komponente sich selbst rendert oder an einen Renderer weiterleitet wird (wie bei der *decode()*-Methode in der Phase

„Apply Request Values“), wird die jeweilige Methode *encodexxx()* der aktuellen UI-Komponente bzw. des Renderers aufgerufen. Zunächst wird die Methode *encodeBegin()* aufgerufen, um am Anfang ein Tag zu rendern. Anschließend wird die Methode *encodeChildren()* aufgerufen, um dessen Kinderobjekten die Möglichkeit zu geben, sich selbst zu rendern; und zum Schluss folgt die Methode *encodeEnd()*, um abschließende Tags der aktuellen Komponente zu rendern.

JSF ermöglicht es dem Entwickler, eigene Renderer sowohl für die bestehenden Komponenten als auch für eigene Komponenten zu schreiben. Dazu muss die neue (Render)-Klasse von der Klasse *javax.faces.render.Renderer* abgeleitet werden und in der Datei *faces-config.xml* einem Render-Kit bekannt gemacht werden (mit der Referenzimplementierung wird ein HTML-RendererKit mitgelie-

fert). Wenn Sie eigene Komponenten entwickelt haben, müssen Sie diese auch in der *faces-config.xml* registrieren lassen (Listing 9).

Das aktuell gerenderte *Tree*-Objekt bleibt im *FacesContext* bestehen und wird von JSF für den aus dieser gerenderten Seite kommenden Request verwendet, um den Standard Request Processing Lifecycle erneut auszuführen.

### Trennen des Verhaltens einer UI-Komponente von ihrem Renderer

Viele Web-Präsentationsframeworks, darunter auch Struts, verwenden die JSP-Custom Tag Library-Eigenschaften, um den Inhalt einer JSP-basierten Seite zu rendern. Im Gegensatz zu diesen Frameworks verwendet JSF Tag Libs, um die UI-Komponenten mit dem jeweiligen Renderer in einer JSP zu kombinieren. Jedes JSF-Tag in der Referenzimplementierung besteht aus zwei Teilen:

#### Listing 4

Die Methode *processAction* eines Action-Listeners

```
public void processAction(ActionEvent event) {
    ...
    if(event.getActionCommand().equals("anmelden")){
        action = boManager.anmelden(user);
    }
    ApplicationFactory factory = (ApplicationFactory)
    FactoryFinder.getFactory(FactoryFinder.
        APPLICATION_FACTORY);
    factory.getApplication().getNavigationHandler().
    handleNavigation(FacesContext.getCurrentInstance(),
        null,action.invoke());
    ...
}
```

#### Listing 5

Die UI-Komponente *HREFGraphic*

```
public class HREFGraphic extends UICommand {
    ...
    public boolean getRendersSelf() {
        return false;
    }
    public String getRendererType() {
        return "HREFGraphicRenderer";
    }
}
```

#### Listing 6

Die Renderer-Klasse *HREFGraphicRenderer* für *HREFGraphic*

```
public class HREFGraphicRenderer extends Renderer {
    public boolean supportsComponentType(
        String componentType) {
        if (componentType == null) {
            throw new NullPointerException();
        }
        return (componentType.equals(HREFGraphic.TYPE));
    }
    public void encodeBegin(FacesContext context,
        UIComponent component)
        throws IOException {
        if (context == null || component == null) {
            throw new NullPointerException();
        }
    }
    public void encodeChildren(FacesContext context,
        UIComponent component)
        throws IOException {
        if (context == null || component == null) {
            throw new NullPointerException();
        }
    }
    public void encodeEnd(FacesContext context,
        UIComponent component) throws IOException {
        writer.write("<a href="+"/IAF_ALFA_PRES1_JSFLIENT/
            faces/jsp/kfzSuchergebnis.jsp"+"?index=");
        writer.write(Integer.toString(key.intValue()));
        writer.write(">");
        writer.write(contextPath+(String)hrefGraphic.
            getAttribute("image"));
        writer.write("\ alt="+hrefGraphic.getAttribute
            ("alt")+""");
        writer.write(" onmouseover="+this.src="+contextPath+
            hrefGraphic.getAttribute("image2")+""");
        writer.write(" onmouseout="+this.src="+contextPath+
            hrefGraphic.getAttribute("image")+""");
        writer.write(">");
        writer.write("</a>");
    }
    public void decode(FacesContext context,
        UIComponent component)
        throws IOException {
        counter=0;
        if (context == null) {
            throw new NullPointerException();
        }
        Map requestParameterMap = cotext.getExternal
            Context().getRequestParameterMap();
        String value = (String)requestParameterMap.get
            ("index");
        ...
        Object obj = convertor.getAsObject(value);
        HREFGraphic hrefGraphic = (HREFGraphic)component;
        ((String)hrefGraphic.setIndex(obj);
        hrefGraphic.fireActionEvent(context);
        hrefGraphic.setValid(true);
    }
}
```



Abb. 4: Die Darstellungsformen der Komponente UICommand

Während der erste Teil eine UI-Komponente darstellt, der das Verhalten bzw. den Zustand dieser Komponente auf der Server-Seite repräsentiert, stellt der zweite Teil den jeweiligen Renderer dar. Um das besser zu veranschaulichen, schauen wir uns die Komponente *UICommand* einmal näher an. Abbildung 4 zeigt, wie die Komponente *UICommand* einen Button oder Hyperlink mit Hilfe eines Tags in einer JSP rendern

### Listing 7

Die Klasse *HREFTag*

```
public class HREFTag extends FacesTag {
    ...

    public void overrideProperties
        (UIComponent component) {
        super.overrideProperties(component);
        HREFGraphic hrefGraphic = (HREFGraphic)component;
        if(hrefGraphic.getAttribute("valueRef") == null) {
            hrefGraphic.setAttribute("valueRef", getValueRef());
        }
        if(hrefGraphic.getAttribute("valueRefSelected") ==
            null) {
            hrefGraphic.setAttribute("valueRefSelected",
                getValueRef());
        }
        if(hrefGraphic.getAttribute("action_listener") == null)
        {
            hrefGraphic.setAttribute("action_listener",
                getAction_listener());
        }
        if(hrefGraphic.getAttribute("alt") == null) {
            hrefGraphic.setAttribute("alt", getAlt());
        }
        ...
    }
    // Gibt den Renderer zurück
    public String getRendererType() {
        return "HREFGraphicRenderer";
    }
    // Gibt die Komponente zurück
    public String getComponentType() {
        return ("HREFGraphic");
    }
    ...
}
```

kann. Dadurch haben Sie die Möglichkeit, die bestehenden Komponenten mit einem eigenen Renderer zu erweitern, wenn Sie nur die Darstellung dieser Komponente – zum Beispiel um Skript-Sprachen-Elemente erweitert – ändern möchten. In Listing 5 wird die Methode *getRenderersSelf* der Komponente *HREFGraphic* überschrieben und liefert jetzt *false* zurück. Da der Aufruf von *getRenderersSelf false* zurückliefert, wird die Methode *getRendererType* der Komponente *HREFGraphic* aufgerufen, um den Renderer für diese zu initialisieren und anschließend diese zu rendern.

Da *HREFGraphic* die Rendering-Aufgabe an die Klasse *HREFGraphicRenderer* weiterleitet, müssen wir diese Klasse zur Verfügung stellen (Listing 6). Da *HREFGraphic* keine Kinder-Komponenten hat, implementiert man nur die Methode *encodeEnd*, um diese Komponente zu rendern. In diesem Fall wird ein Hyperlink mit einem Image dargestellt, das sich beim Maus-Eintritt in eine andere Farbe bzw. Image ändert.

Danach soll diese Komponente eine Tag-Klasse implementieren, die die Klasse *javax.faces.webapp.FacesTag* ableiten muss. In *HREFTag* wird *HREFGraphic* mit dem Renderer *HREFGraphicRenderer* kombiniert. Die *overrideProperties*-Methode der Klasse *HREFTag* aktualisiert die *HREFGraphic*-Werte mit den in der JSP definierten Werten (Listings 7 bis 9).

*getRendererType* und *getComponentType* liefern jeweils mit diesem Tag referenzierte Komponenten bzw. Renderer zurück (Listing 7). Selbst entwickelte Komponenten und Renderer müssen in der *faces-config.xml* am Renderer-Kit bzw. JSF registriert werden (Listing 8). Listing 9 zeigt die UI-Komponente *HREFGraphic* mit dem Renderer *HREFGraphicRenderer* in einer JSP kombiniert dargestellt.

### JSF Navigation Model

Das JSF Navigation Model ähnelt in seinem Grundzügen dem Struts Navigation Model: Man definiert die Navigationsregeln im Applikationskonfigurationsfile *faces-config.xml*. Eine Navigationsregel legt fest, wie man von einer bestimmten Seite zu anderen Seiten wechseln kann. Diese Regeln werden mit dem Element `<navigation-rule>...</navigation-rule>`

## Anzeige

### Listing 8

```
<faces-config>
...
<render-kit>
  <renderer>
    <renderer-type>HREFGraphicRenderer
    </renderer-type>
  </render-kit>
  <renderer-class>
    de.wwag.arch.alfa.presentation.renderer.
    HREFGraphicRenderer
  </renderer-class>
</render-kit>
<component>
  <component-type>HREFGraphic</component-type>
  <component-class>
    de.wwag.arch.alfa.presentation.components.
    HREFGraphic
  </component-class>
</component>
...
</faces-config>
```

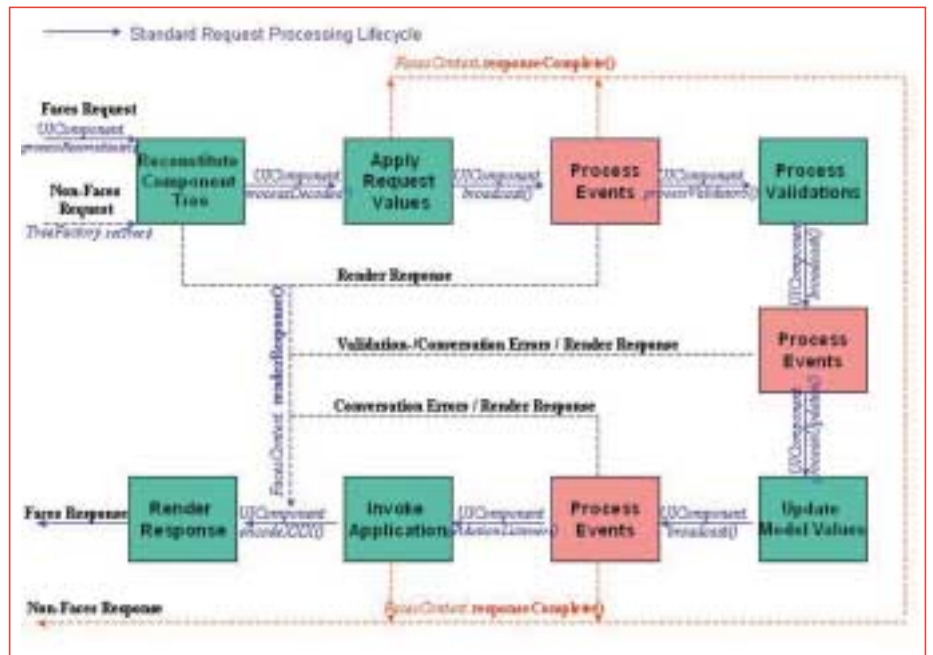


Abb. 5: JSF Lifecycle

### Listing 9

Das Tag `hrefgraph` in einer JSP

```
<ww:hrefgraph_hymlink id="test"
valueRef="ablaufForm.fahrzeugsucheBean.
parameterFahrzeuge"
index="index"
valueRefSelected="ablaufForm.fahrzeugsucheBean.
selectedIndex" alt="Uebernehmen.."
image="/images/uebernb.gif" image2="/images/
ueberna.gif">
<f:action_listener type=
"de.wwag.arch.alfa.presentation.jsfClient.actions.
HREFGraphicActionListener"/>
</ww:hrefgraph_hymlink>
```

### Listing 10

Auszug aus der Datei `faces-config.xml`

```
<faces-config>
...
<navigation-rule>
  <from-tree-id>/jsp/anmelden.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/jsp/pkwSucheingabe.jsp</to-tree-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failed</from-outcome>
    <to-tree-id>/jsp/anmeldung.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
...
</faces-config>
```

definiert. Die Auswahl der Navigationsregeln ist abhängig von der aktuellen Seite. Das bedeutet, dass die Navigationsregeln auf die Seiten bezogen sind und pro Seite nur eine Navigationsregel zu definieren ist (Listing 10).

### Listing 12

Die Komponente `command_button` mit dem Attribut `action`

```
<h:command_button id="command_an" key="anmelden"
bundle="PR_Bundle" commandName="anmelden"
action="success">
</h:command_button>
```

### Listing 11

Die Methode `anmelden` aus der JavaBean-Klasse `AblaufForm`

```
public Action anmelden(){

return new Action(){
public String invoke(){
String result=null;
try{
AnmeldungsManager.getInstance().anmelden();
result="success"
}catch(Exception e){
result="failed"
}
}
};
}
```

Das Element `<from-tree-id>...</from-tree-id>` definiert die jeweilige Seite. Eine Navigationsregel kann mehrere Navigationsfälle haben, die durch das Element `<navigation-case>...</navigation-case>` definiert werden. Jeder Navigationsfall hat ein `<from-outcome>...</from-outcome>`-Tag und ein `<to-tree-id>...</to-tree-id>`-Tag. In `<from-outcome>...</from-outcome>` wird der Text – z.B. success, failed usw. – eingegeben, der dann durch den Aufruf der Methode `invoke` der jeweiligen Action zurückgeliefert wird. Wenn wir die Button-Komponente in der Datei `anmeldung.jsp` wie in Listing 3 ändern, wird beim Klicken dieses Buttons der Standard `ActionListener` aufgerufen.

Das Attribut `actionRef` ist eine Referenz auf eine Methode einer Bean (meistens des Modells). JSF ruft die in dem Attribut `actionRef` definierte Methode auf, die ein Objekt von der Klasse `Action` zurückliefert. Anschließend wird die Methode `invoke` des zurückgelieferten `Action`-Objektes aufgerufen. Diese Methode liefert den Text zurück, den man in einem Navigationsfall `<navigationcase>...</navigation-case>` in dem Tag `<from-outcome>...</from-outcome>` definiert hat (Listings 10-11). In unserem Fall wird, wenn der zurückgelieferte Text success ist, aus der in `<to-tree-id>...</to-tree-id>` eingege-

benen Seite */jsp/pkwSucheingabe.jsp* ein *Tree*-Objekt erzeugt und in dem aktuellen *FacesContext* ablegt, um gerendert zu werden.

Es gibt auch die Möglichkeit, über das Attribut *action* zu definieren, was in den jeweiligen Navigationsregeln mit dem Attribut *<from-outcome>...</from-outcome>* gemappt werden soll (Listing 12). Danach wird aus der in *<to-tree-id>...</to-tree-id>* definierte JSP ein *Tree*-Objekt erzeugt und im aktuellen *FacesContext* ablegt, um gerendert zu werden. Dadurch haben die Entwickler die Möglichkeit, die JSP zu testen, ohne eine Businesslogik-Methode aufzurufen.

### Event Handling

Das JSF Event-Konzept ähnelt dem der JavaBeans bzw. des Swing Event-Konzepts. Für jedes *XXXEvent* existiert ein entsprechender *XXXListener*, den man implementieren muss, um auf das jeweilige Event reagieren zu können. Dieser Listener wird für das zu beobachtende Objekt mit Hilfe einer Anmelde-methode registriert. Daher wird es einem Java-Programmierer nicht schwer fallen, auf das JSF Event-Konzept umzusteigen. In JSF werden zwei Arten von Events unterstützt, eines davon ist das *javax.faces.event.ActionEvent*, das von einer *UI-Command*-Komponente ausgelöst werden kann. Wie im Java Event-Konzept gibt es für jedes Event als Gegenstück einen Listener. Diese Klassen, die das Interface *javax.faces.event.ActionListener* implementieren, müssen sich bei einer *UI-Command*-Komponente mittels JSF Tag Libs registrieren, um auf die von dieser Komponente ausgelösten Events zu reagieren.

Das andere Event, das JSF unterstützt, ist *javax.faces.event.ValueChangeEvent*. Das *ValueChangeEvent*-Objekt beinhaltet den alten und neuen Wert der Komponente, die dieses Event ausgelöst hat. Zum Beispiel kann die Komponente *selectboolean\_checkbox* so ein Event auslösen, wenn man sie ausgewählt hat. Im folgenden Codeabschnitt wird der Listener *AlfaValueChangeListener* für die Komponente *selectboolean\_checkbox* registriert. Wenn man die Checkbox auswählt, wird die Methode *processValueChanged* von diesem Listener aufgerufen:

```
<h:selectboolean_checkbox id="gesch" title="Geschlecht"
    valueRef="myModel.geschlechtSelected">
    <f:valuechanged_listener
        type="de.wwag.arch.alfa.presentation.jsfClient.
            actions.AlfaValueChangeListener"/>
</h:selectboolean_checkbox>
```

Diese beiden Interfaces sind von dem Interface *javax.faces.event.FacesListener* abgeleitet, das die wichtige Methode *getPhaseId()* deklariert. Diese Methode liefert eine *PhaseIdentifikation* zurück, welche die Phase beschreibt, nach deren Ausführung der Listener aufgerufen werden muss. Liefert die Methode *getPhaseId()* eine *PhaseId.ANY\_PHASE* zurück, wird

dieser Listener in jeder „Process Events“-Phase aufgerufen, sobald ein Event ausgelöst wird (Abb. 5).

### Erzeugen einer Anwendungsmodell JavaBean

Für jede Komponente, die ein Attribut *valueRef* in einer JSP besitzt, muss in dem referenzierten Modell nach der JavaBeans-Richtlinie eine *getXXX-* und *setXXX-* Methode implementiert sein, d.h. neben der entsprechenden Variablen muss auch noch eine *getter-* und *setter-*Methode existieren. Listing 13 zeigt die Attribute bzw. die Methoden der Klasse *de.wwag.arch.alfa*.

## Anzeige

*presentation.formbeans.AblaufForm* für die UI-Komponente `<h:input_text id="input_bk" valueRef="ablaufForm.benutzerkennung"/>`. Alle nach JavaBeans-Richtlinien entwickelten Klassen können als Modell verwendet werden, ohne dass sie vorher von einer JSF-Klasse abgeleitet werden müssen.

### Listing 13

#### Das JSF-Modell

```
public class AblaufForm{
    private String benutzerkennung;
    ...
    public void setBenutzerkennung
        (String benutzerkennung) {
        this.benutzerkennung = benutzerkennung;
    }
    public String getBenutzerkennung () {
        return benutzerkennung;
    }
}
```

### Listing 14

#### Die Konfigurationsdatei *faces-config.xml* von JSF

```
<faces-config>
...
<managed-bean>
    <managed-bean-name>ablaufForm
        </managed-bean-name>
    <managed-bean-class>
        de.wwag.arch.alfa.presentation.formbeans.
            AblaufForm
    </managed-bean-class>
    <managed-property>
    <property-name>applicationDataBean</property-name>
    <value-ref>applicationDataBean</value-ref>
    </managed-property>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>boManager
        </managed-bean-name>
    <managed-bean-class>
        de.wwag.arch.alfa.presentation.businesslogic.BOManager
    </managed-bean-class>
    <managed-property>
    <property-name>ablaufForm</property-name>
    <value-ref>ablaufForm</value-ref>
    </managed-property>
    <managed-bean-scope>session
        </managed-bean-scope>
</managed-bean>
...
</faces-config>
```

Eine weitere Neuheit der JSF ist, dass die Initialisierung von JavaBeans in der Datei *faces-config.xml* definiert werden kann. Man hat dabei die Möglichkeit, über ein Element `<value-ref>...</value-ref>` von einem Modell auf ein anderes zu referenzieren bzw. die Attribute des Modells zu initialisieren. Der Alias-Name *ablaufForm* kann in den JSPs mit dem Attribut *valueRef* referenziert werden (Listing 14). Es ist hier zu beachten, dass das referenzierte Objekt mindestens die gleiche Lebensdauer haben soll, wie das anzeigende Objekt (zum Beispiel ein Objekt im Scope „session“ kann nur auf die Objek-

String {} zu definieren, der während der Laufzeit ersetzt wird.

### Fazit

Die JavaServer Faces-Technologie bringt das MVC-Konzept einen guten Schritt weiter, indem die View für die Web-Anwendung auf der Server-Seite in zwei Komponenten geteilt wird. Eine von diesen Komponenten ist die *UIComponent*, die für das Verhalten und den Zustand zuständig ist. Die andere ist der Renderer, der die jeweilige UI-Komponente darstellt. Die Trennung des Verhaltens bzw. Zustandes einer Komponente vom eigenen Renderer ist ein überaus nützlicher Ansatz. Er ermöglicht Entwicklern die Wiederverwendung von UI-Komponenten unabhängig vom Renderer und versetzt sie somit in die Lage, schnell und einfach multikanalfähige Anwendungen zu entwickeln.

Obwohl die JSF noch nicht freigegeben wurden, kann man bereits mit der JSF-Referenzimplementierung recht viel erreichen. Zusätzlich zu ihrer sauberen Architektur werden die JSF eine Standard-schnittstelle von Java ab der J2EE 1.4 sein.

Allein das Wort „Standard“, worauf man seit langer Zeit wartet, macht das Framework für Entwickler sehr interessant. Obwohl JSF im Vergleich zu anderen Frameworks etwas komplexer ist, wird sich die Einarbeitung in JSF für Entwickler langfristig auf jeden Fall lohnen. Darüber hinaus findet man auch als Java-Kenner altbekannte Konzepte sowie z.B. Eventhandling, Rendering und Internationalisierung wieder. Die Verwendung des Composite Design-Patterns bietet ein hohes Maß an Flexibilität, was auf der anderen Seite allerdings durch eine geringere Performance bezahlt werden muss. Keine Rose ohne Dornen! ■

### Links & Literatur

- Kruchten, Philippe: „The Rational Unified Process“, Addison-Wesley, 1998
- Gamma, Helm, Johnson, Vlissides: „Design Patterns“, Addison-Wesley, 1997
- JavaServer Faces (JSF): [java.sun.com/j2ee/javaserverfaces/](http://java.sun.com/j2ee/javaserverfaces/)
- Öztürk, Muhammet: „Face Shop – Beispielprojekt mit JavaServer Faces (JSF)“, *Java Magazin* 1.2004
- Öztürk, Muhammet: „Fliegen in der Nacht – das Webframework Struts“, *Java Magazin* 4.2002

## Mehrsprachige Web-Anwendungen

te im Scope „session“ oder „application“ zeigen). Passiert dies nicht, wird ein *NullPointerException* geworfen, da das referenzierte Objekt bereits gelöscht wurde.

### Internationalisierung

JSF bietet für die Internationalisierung Ihrer Web-Anwendungen ein sehr einfaches Konzept, so genannte Resource-Bundles, um mehrsprachige Web-Anwendung zu entwickeln. Das Konzept basiert auf der bekannten Java-Internationalisierung. Dabei wird für das jeweilige Land eine *ApplicationResources\_xx.properties*-Datei erstellt, die die Text-einträge mit den entsprechenden Key/Value-Paaren beinhaltet. Die Zeichen *xx* im Dateinamen bedeuten die Länderkürzel, für Deutschland wird z.B. der Dateiname *ApplicationResources\_de.properties*. Wie in Listing 1 zu sehen ist, verwendet JSF auch das JSTL Internationalization Tag `<fmt:setBundle>` für die Lokalisierung dieser Properties-Datei. JSF ruft den zu dem Key gehörenden Wert mittels der Attribute *key* und *bundle* ab. Zum Beispiel wird die Komponente `<h:output_text id="output_kw" key="anmeldung.kennwort" bundle="PR_Bundle"/>` mit dem Wert ersetzt, der zum *key="anmeldung.kennwort"* gehört. JSF unterstützt auch das *java.text.MessageFormat*, das es erlaubt, Platzhalter mit dem

## Online-Shop – Beispielprojekt mit JavaServer Faces

von Muhammet Öztürk und Marco Michel

# Face Shop

Im ersten Teil unseres Titelthemas auf Seite 38 wurden die Konzepte von JSF beschrieben und ein Grundlagenwissen vermittelt. Um die Konzepte der JSF zu veranschaulichen, haben wir beispielhaft einen Online-Shop erstellt. Im Folgenden werden wir das Programm und die Code-Ausschnitte vorstellen und dabei JSF in der Praxis zeigen.

### Der Online-Shop

Der Online-Shop soll ermöglichen, dass verschiedene Artikel, aufgeteilt in mehrere Gruppen, in der Geschäftslogikschicht (repräsentiert durch den Business Delegate) abgerufen werden können. Um das Beispiel nicht zu komplex werden zu lassen, haben wir auf die Persistenzschicht verzichtet. Der Business Delegate instanziiert und hält die Artikel, mit welchen direkt gearbeitet wird. Da zur Vereinfachung das Objekt *Artikel* auch die vom Anwender gewünschte Menge speichert, muss es für jeden User eine Instanz pro Artikel geben. Dies wird erreicht, indem es pro Session einen Business Delegate gibt.

Wie in Abbildung 1 zu erkennen ist, gibt es eine klare Trennung zwischen View (JSP), Modell, Controller (*Action Manager* und *XXAction*) und der Businesslogik (Business Delegate). Es wird also sauber zwischen den Daten, der Präsentationslogik und der Geschäftslogik getrennt. Um diese Trennung zu erreichen, wurde der Zugriff auf die JSF-Actions nicht wie im JSF-Tutorial beschrieben als *get*-Methode im Modell, sondern in einem Action Manager realisiert. Dieser verwaltet alle Actions und stellt sie zur Verfügung. In einer JSP verweist jede *actionRef* (Referenz auf eine Action für die

Navigation) auf den Action Manager, jede *valueRef* (Referenz auf einen Wert) auf das Modell. Das Modell beinhaltet dabei die ausgewählte Artikelgruppe, die aktuell angezeigten Artikel, ein Kundenobjekt sowie den Warenkorb. Auf Internationalisierung wurde in diesem Beispiel verzichtet. Der Shop bietet folgende einfache Funktionen:

- Auswahl einer Artikelgruppe und Anzeige der entsprechenden Artikel,
- Mengenangabe pro Artikel,
- Aktualisierung des Warenkorbes,
- Anzeige des Warenkorbes,
- Eingabe von Kundendaten und
- Einkaufen der ausgewählten Artikel.

### Erzeugen von JavaBeans

Die Klassen für den Action Manager, das Modell und den Business Delegate werden im JSF-Konfigurationsfile *faces-config.xml* verwaltet. Wie zuvor im theoretischen Teil dieses Schwerpunkts beschrieben, können hier Klassen verwaltet, ihre Lebensdauer

angegeben und Referenzen auf andere Klassen zugewiesen werden. Alle Klassen bekommen im Beispiel den Scope-Session. Der Action Manager erhält zudem über das *<value-ref>*-Element eine Referenz auf das Modell und den Business Delegate. Der entsprechende Code-Ausschnitt ist in Listing 1 zu sehen.

Da JSF hier mit den JavaBeans-Konzepten arbeitet, müssen der Action Manager die Attribute *model* und *delegate* mit den entsprechenden *Getter*- und *Setter*-Methoden haben, damit JSF die im Config-File angegebenen Referenzen setzen kann.

Wie schon erwähnt, verwaltet der Action Manager die Actions und stellt sie den JSPs über *getter*-Methoden zur Verfügung. Beim Anlegen einer Action wird dieser die Referenz auf das Modell und den Business Delegate mitgegeben. Somit kann die Action auf die Geschäftslogik zugreifen und das Modell verändern. Jede Action ist in einem eigenen Class-File implementiert. Das bewirkt, dass man diese

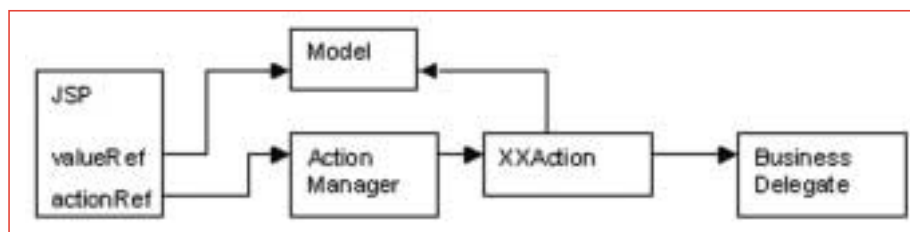


Abb. 1: Schematischer Aufbau des Online-Shops

Klasse erstens in anderen Projekten als Komponente verwenden kann und zweitens vereinfacht es die Arbeit in einem Team und verhindert konkurrierende Zugriffe, falls mehrere Entwickler verschiedene Actions anpassen müssen. In Listing 2 sind Code-Fragmente aus dem Action Manager zu sehen.

### Eigene Komponente *UItemCount*

Das Modell des JSF-Shops beinhaltet eine Liste mit den aktuell angezeigten Artikeln. Wie bereits erwähnt, beinhaltet jeder Artikel ein Attribut mit der Mengenangabe des Users. Diese Liste muss im Browser angezeigt werden und zu jedem Artikel soll ein Eingabefeld für die Mengenangabe zur Verfügung stehen. Die Standard-Komponenten von JSF brauchen eine eindeutige

Zuordnung von Eingabefeld zu Modell-Attribut. Eine Liste im Modell kann zwar angezeigt, aber nicht mehr aktualisiert werden, da die Zuordnung zwischen Eingabefeld und Listeneintrag nicht funktioniert. Dies ist aber nötig, um die Mengenangaben des Users in der Artikelliste zu aktualisieren.

Um diese Anforderung zu realisieren, hat der Shop eine eigene Komponente *UItemCount*, welche die Mengenangaben aus der Artikelliste beinhaltet und sie aktualisieren kann. Da es sich hierbei um eine Eingabekomponente handelt, wird die Klasse *javax.faces.component.UIInput* als Basisklasse genommen. Zudem gibt es für diese Komponente einen Renderer, welcher pro Artikel in der Liste ein Textfeld anzeigt, in welchem der User die gewünschte Menge des Artikels angeben kann. Des Weiteren muss, um die Komponente verwenden zu können, ein JSP-Tag entwickelt werden. Der Renderer kann aber nicht in einem einzigen Durchlauf alle Textfelder rendern, da er die Anordnung

der einzelnen Textfelder im Browser nicht kennt. Zudem werden evtl. weitere Informationen pro Artikel wie Name und Beschreibung angezeigt. Die Aufbereitung der Informationen und die Anordnung in einer Tabelle erfolgt in der JSP-Seite. Folglich wird auch die Schleife in der JSP programmiert. Listing 3 zeigt, wie mit den JSF-Tags eine Tabelle gebaut und die Inhalte einer Liste angezeigt werden. Hier wird auch das eigene Tag *<itemcount\_text>* eingesetzt, welches auf den Renderer verweist. Das Tag hat ein Referenz-Attribut auf die Artikelliste im Modell. Hieraus ergibt sich, dass der Renderer immer nur ein einzelnes Textfeld rendern darf.

Damit der Renderer bei jedem Durchlauf ein einzelnes Textfeld anzeigen kann und dabei die Liste der Artikel durchgeht (es müssen ja die bereits gespeicherten Mengenangaben angezeigt werden), hat die Komponente einen internen Zähler, welcher bei jedem Durchlauf hochgezählt wird. Der Zähler muss als Instanzvariable definiert werden, damit er nicht bei jedem Aufruf der *encode*-Methode neu initialisiert wird. Allerdings kann der Zähler nicht im Renderer definiert werden, da dieser als Singleton existiert und somit keine Multisession-Fähigkeit mehr gegeben wäre. Von der Komponente gibt es aber für jeden Request eine neue Instanz, folglich wurde hier der Zähler definiert (Listing 4).

#### Listing 1

```
<managed-bean>
<description>BusinessDelegate fuer das JSF-Shopping.
</description>
<managed-bean-name>businessdelegate
</managed-bean-name>
<managed-bean-class>shopping.delegate.
BusinessDelegateImpl</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
<description>Modell fuer das JSF-Shopping.</description>
<managed-bean-name>shoppingbean
</managed-bean-name>
<managed-bean-class>shopping.model.ShoppingModel
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
<description>ActionManager fuer das JSF-Shopping.
</description>
<managed-bean-name>actionmanager
</managed-bean-name>
<managed-bean-class>shopping.action.
ActionManager</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
<managed-property>
<property-name>model</property-name>
<value-ref>shoppingbean</value-ref>
</managed-property>
<managed-property>
<property-name>delegate</property-name>
<value-ref>businessdelegate</value-ref>
</managed-property>
</managed-bean>
```

#### Listing 2

```
private MenuAction menuAction = null;
/**
 * Liefert eine MenuAction
 */
public Action getMenuAction()
{
    if (menuAction == null) menuAction = new MenuAction
        (model, delegate);

    return menuAction;
}
/**
 * Das Modell des Shops
 */
private ShoppingModel model;
/**
 * Returns the model.
 * @return ShoppingModel
 */
public ShoppingModel getModel()
{
    return model;
}
/**
 * Sets the model.
 * @param model The model to set
 */
public void setModel(ShoppingModel model)
{
    this.model = model;
}
```

#### Listing 3

```
<h:panel_list id="gruppe">
<f:facet name="header">
<h:panel_group>
<h:output_text value="Titel"/>
<h:output_text value="Beschreibung"/>
<h:output_text value="Preis"/>
<h:output_text value="Menge"/>
</h:panel_group>
</f:facet>
<h:panel_data id="gruppeData" var="gruppeRow"
valueRef="shoppingbean.currentArticles">
<h:output_text id="title" valueRef="gruppeRow.title"/>
<h:output_text id="desc" valueRef="gruppeRow.
description"/>
<h:output_text id="price" valueRef="gruppeRow.price"/>
<s:itemcount_text id="itemcount" valueRef=
"shoppingbean.currentArticles" size="4"/>
</h:panel_data>
</h:panel_list>
```

Eine Komponente bzw. ihre Renderer besitzt eine *encode*- und eine *decode*-Methode. Der Renderer muss zudem angeben, welchen Komponententyp er unterstützt (Listing 5).

Die *decode*-Methode des Renderers wird in der *Apply Request Values Phase* des JSF-Lifecycles aufgerufen. Sie ermöglicht jeder Komponente, ihre Werte aus dem Request zu holen. Die Request-Para-

meter werden dann in das interne Format der Komponente decodiert (Listing 6).

Dieser Renderer holt sich zunächst einen String-Array mit allen Werten der Textfelder aus dem Request. Da alle Ein-

#### Listing 4

```
/**
 * interner Zähler für den Index in der Liste der Artikel
 */
private int ind = 0;
/**
 * Returns the ind.
 * @return int
 */
public int getInd()
{
    return ind;
}
/**
 * Gibt den Zähler zurück und erhöht ihn anschließend
 * um eins.
 * @return int
 */
public int increaseInd()
{
    return ind++;
}
/**
 * Sets the ind.
 * @param ind The ind to set
 */
public void setInd(int ind)
{
    this.ind = ind;
}
```

#### Listing 5

```
public boolean supportsComponentType
    (String componentType)
{
    if (componentType == null)
    {
        throw new NullPointerException();
    }
    return (componentType.equals(UiItemCount.TYPE));
}
```

#### Listing 6

##### decode-Methode des Renderers

```
public void decode(FacesContext context,
    UIComponent component)
    throws IOException
{
    if (context == null || component == null)
    {
        throw new NullPointerException();
    }

    UiItemCount itemCount = (UiItemCount) component;
    // Flag ob die Eingabe valid war
    boolean valid = true;
    // den internen Zähler wieder auf 0 setzen
    itemCount.setInd(0);
    // die Parameter aus dem Request holen
    Map requestParameterMap = context.getExternalContext().
        getRequestParameterValuesMap();
    String[] valueList = (String[])requestParameterMap.get
        (itemCount.getAttribute(UiItemCount.ID));

    // Wenn es Eingabewerte gibt
    if (valueList != null)
    {
        // einen Array für die Anzahl der Artikel anlegen
        int[] counts = new int[valueList.length];

        // die einzelnen Eingabewerte durchgehen
        for (int i=0; i<valueList.length; i++)
        {
            try
            {
                // den Eingabewert in einen int umwandeln und in den
                // neuen Array ablegen
                counts[i] = Integer.parseInt(valueList[i]);
            }
            catch (NumberFormatException nfe)
            {
                // wenn das nicht geht, war die Eingabe falsch
                valid = false;
            }
        }
    }

    // die Eingabe ist auch dann falsch,
    // wenn der Wert kleiner als 0 ist
    if (counts[i] < 0) valid = false;
}

if (valid)
{
    // bei richtiger Eingabe den int-Array mit der Menge
    // ablegen
    itemCount.setAttribute
        (UiItemCount.ITEM_COUNTS_ATTR, counts);
}
else
{
    // bei falscher Eingabe die ursprünglichen Eingabewerte
    // ablegen um sie nochmal zur Korrektur anzeigen zu könne
    itemCount.setAttribute(UiItemCount.ITEM_COUNTS_ATTR,
        valueList);
    // und einen Fehler text im Context ablegen
    Message errMsg = Resources.getMessageResources().
        getMessage(context, UiItemCount.COUNT_
            INVALID_MESSAGE_ID);

    context.addMessage(itemCount, errMsg);
}

// jetzt noch bekanntgeben, ob alles korrekt war
itemCount.setValid(valid);

requestParameterMap = null;
valueList = null;
}
```

## Anzeige

gabefelder für die Mengenangaben den gleichen Namen haben, muss dazu die Methode `getRequestParameterValuesMap()`. `get(String)` aufgerufen werden. Anschlie-

ßend wird ein *int*-Array angelegt, die einzelnen Werte decodiert und darin gespeichert. Geht dies schief, wird ein entsprechender Fehler im *FacesContext* abgelegt und die eingegebenen fehlerhaften Werte in der Komponente gespeichert. Wenn alles geklappt hat, wird der decodierte *int*-Array in der Komponente gespeichert. Abschließend muss der Komponente noch bekannt gegeben werden, ob die *decode*-Methode erfolgreich war.

Die *encode*-Methoden des Renderers werden in der *Render Response Phase* des JSF-Lifecycles aufgerufen. Hier werden die Komponenten gerendert. Da wir keine Kind-Komponenten haben, brauchen wir nur die *encodeEnd*-Methode auszuimplementieren (Listing 7).

Diese Methode wird für jede Zeile in unserer JSP-Tabelle, also für jeden einzelnen Artikel in der Liste, aufgerufen. Zunächst prüft die Methode, ob es in der



Abb. 2: Auswahl einer Artikelgruppe

Komponente einen Array mit „alten“ Eingabewerten gibt. Dieser wird, wie wir oben gesehen haben, im Fehlerfall von der *decode*-Methode angelegt. Im Falle einer fehlerhaften Eingabe wird die gleiche Seite nochmals angezeigt, um dem User die Möglichkeit zu geben, die eingegebenen Daten zu korrigieren. Dazu müssen die alten, fehlerhaften Eingabewerte angezeigt werden. Wenn es Eingabewerte gibt, wird ein Eintrag aus dem Array als *Value*-Wert für das Eingabefeld genommen. Der interne Zähler aus der Komponente verweist dabei auf den aktuellen Eintrag und wird anschließend um 1 erhöht.

Gibt es keine Eingabewerte, holt sich die Methode die Daten aus dem Modell,

### Listing 7

```
public void encodeEnd(FacesContext context,
                    UIComponent component)
throws IOException
{
    if (context == null || component == null)
    {
        throw new NullPointerException();
    }

    UiItemCount itemCount = (UiItemCount) component;
    // die Liste der zuvor eingegebenen Werte
    Object[] counts = (Object[]) itemCount.getAttribute(
        (UiItemCount.ITEM_COUNTS_ATTR));

    // Value-Wert für das Textfeld
    Object value = "";

    // Wurden zuvor Werte eingegeben, zeigen wir wieder diese
    // an (z.B bei einer fehlerhaften Einagbe)
    if (counts != null)
    {
        value = counts[itemCount.increaseInd()];
    }
    // ansonsten holen wir die Werte aus dem Modell
    else
    {
        // die Liste der aktuellen Artikel vom Modell holen
        List currentArticles = (List)
            (Util.getValueBinding(itemCount.getValueRef()))
                .getValue(context);

        // Der Wert im Textfeld ist die Menge des Artikels
        Article article = (Article) currentArticles.get(
            itemCount.increaseInd());
        value = Integer.toString(article.getCount());
        currentArticles = null;
        article = null;
    }

    // wenn im Tag eine Grösse angegeben wurde,
    // nehmen wir diese
    String size = "";
    if (itemCount.getAttribute(UiItemCount.SIZE) != null)
        size = " size=\"" + itemCount.getAttribute(
            (UiItemCount.SIZE)) + "\"";

    // jetzt geben wir das Textfeld aus
    ResponseWriter writer = context.getResponseWriter();
    writer.write("<input type=\"text\" name=\"" +
        itemCount.getAttribute(UiItemCount.ID) +
        "\" value=\"" + value + "\" " + size);
    writer.write(">");
    writer = null;
}
```

### Listing 8

```
updateModel-Methode der UiItemCount-Komponente
public void updateModel(FacesContext context)
{
    // wenn die Komponentendaten nicht valid sind,
    // wird nichts gemacht
    if (!isValid()) return;

    // wenn kein Modell bekannt ist, wird auch nichts gemacht
    if (getValueRef() == null) return;

    // das Modell holen
    List currentArticles = (List) (Util.getValueBinding(
        (getValueRef())).getValue(context);

    // und die Daten der Komponente im Modell setzen
    int[] itemCounts = (int[]) getAttribute(
        (ITEM_COUNTS_ATTR));

    Article article = null;

    for (int i=0; i<itemCounts.length; i++)
    {
        article = (Article) currentArticles.get(i);
        article.setCount(itemCounts[i]);
    }

    article = null;
    currentArticles = null;
    itemCounts = null;

    // hat alles geklappt, also die lokalen Daten entfernen
    setAttribute(ITEM_COUNTS_ATTR, null);
    setValid(true);
}
```

### Listing 9

```
RecalcAction-Listener (Aktualisierung Warenkorb)
public class RecalcActionListener implements
    ActionListener
{
    public PhaseId getPhaseId()
    {
        return PhaseId.UPDATE_MODEL_VALUES;
    }

    public void processAction(ActionEvent event)
    {
        // den Context holen
        FacesContext context = FacesContext.
            getCurrentInstance();

        // das Modell holen
        ShoppingModel model = (ShoppingModel)
            (Util.getValueBinding(Resources.SHOPPING_
                BEAN_NAME)).getValue(context);

        // den Warenkorb mit den aktuellen Artikeln
        // aktualisieren
        model.getBasket().updateBasket(model.
            getCurrentArticles());
    }
}
```

auf die über das *valueRef*-Attribut des JSP-Tags verwiesen wird. Dabei verweist wieder der interne Zähler auf den aktuellen Eintrag in der Liste. Die Menge des Artikels ist dann der *Value*-Wert für das Eingabefeld.

Abschließend wird noch geprüft, ob es eine Größenangabe für das Textfeld im JSP-Tag gibt und das Textfeld mit dem zuvor ermittelten *Value*-Wert in die Response geschrieben.

Der JSF-Lifecycle durchläuft verschiedene Phasen. Zwei Phasen haben wir im Renderer kennen gelernt. Bisher sind die eingegebenen Werte aber nur in der Komponente gespeichert, das Modell wurde noch nicht aktualisiert. Dies erfolgt in der *Update Model Values Phase*. Hierfür wird die *updateModel*-Methode der Komponente aufgerufen (Listing 8).

In dieser Methode muss zunächst geprüft werden, ob die Daten gültig sind und ob es eine *valueRef* auf ein Modell gibt. Sind die Bedingungen erfüllt, kann das Modell aktualisiert werden. Unsere Kom-

ponente geht hierfür die Liste der Artikel im Modell durch und setzt die *int*-Werte der Komponente in die jeweiligen Artikel. Abschließend werden die internen Werte der Komponente gelöscht und der Komponente bekannt gegeben, dass die Methode erfolgreich war.

### Event, Aktualisierung Warenkorb

Wenn der Anwender nun neue Mengenangaben zu dem Artikel macht, muss der Warenkorb aktualisiert werden. Diese Aktualisierung kann an verschiedenen Stellen der Anwendung vorkommen, dabei muss nicht unbedingt auf eine neue Seite navigiert werden. Das JSF-Tutorial empfiehlt (und in der Praxis hat sich diese Empfehlung bestätigt) für Events ohne Seitennavigation Listener zu verwenden. Wenn auf eine neue Seite navigiert wird, sollten Actions verwendet werden.

Für die Neuberechnung des Warenkorbes stellt die Anwendung einen *ActionListener* zur Verfügung. Dieser ruft die *Update*-Methode des Warenkorbs

auf und übergibt als Parameter die aktuell angezeigten Artikel, welche die Mengenangaben des Users beinhalten. Um zu gewährleisten, dass der Listener mit aktuellen Daten arbeitet, darf er erst aufgerufen werden, nachdem die Komponente das Modell aktualisiert hat. Dies wird erreicht, indem die *PhaseId.UPDATE\_MODEL\_VALUES* im Listener angegeben wird (Listing 9). Die *PhaseId* gibt an, in welcher Phase des JSF-Lifecycles der Listener aufgerufen werden soll. Unsere eigene Komponente *UiItemCount* aktualisiert also zunächst das Modell mit den Mengenangaben des Users, wenn die Eingaben korrekt waren und aktualisiert im Anschluss den Warenkorb durch den Listener. Waren die Eingaben nicht korrekt, wird die Phase *Update Model Values* beschrieben nicht erreicht und somit auch unser Listener nicht aufgerufen. Dieser Mechanismus der JSF stellt sicher, dass der Warenkorb nur mit korrekten Werten über den Listener aktualisiert wird.

## Anzeige



Abb. 3: Anzeige und Mengenangabe der Artikel



Abb. 4: Anzeige des Warenkorbes

Der Button zur Aktualisierung des Warenkorbes muss also in der JSP den entsprechenden *ActionListener* registrieren. Beim Klick auf diesen Button wird das Modell durch die *UiItemCount*-Komponente aktualisiert, der *RecalcActionListener* aufgerufen, der Warenkorb aktualisiert und die Seite erneut angezeigt.

### Listing 10

#### Navigation-Rule aus der *faces-config.xml*

```
<navigation-rule>
  <from-tree-id>/frontstore.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>store</from-outcome>
    <to-tree-id>/store.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
```

### Listing 11

#### Menu-Action

```
public class MenuAction extends Action
{
  private ShoppingModel model;
  private BusinessDelegate delegate;

  public MenuAction(ShoppingModel model,
                   BusinessDelegate delegate)
  {
    this.model = model;
    this.delegate = delegate;
  }

  public String invoke()
  {
    // die aktuelle Artikelgruppe
    String group = (String) model.getCurrentArticleGroup();
    // Artikelgruppe holen und als aktuelle Artikel setzen
    model.setCurrentArticles(delegate.getArticle(group));
    return "store";
  }
}
```

```
<h:command_button id="recalc" label="neu Berechnen"
                  commandName="recalc">
  <f:action_listener type="shopping.action.
                    RecalcActionListener" />
</h:command_button>
```

### Navigation mit Actions

Bisher haben wir gesehen, wie Beans angelegt und initialisiert werden, wie wir eigene Komponenten erzeugen, wir haben auch einige Phasen des JSF-Lifecycles gesehen und wie wir mit Listnern auf Events reagieren können. Der klassische Fall besteht jedoch darin, dass ein Button gedrückt wird, der Request verarbeitet werden muss und auf eine neue Seite navigiert wird. Dazu müssen die Navigationsregeln in der *faces-config.xml* definiert werden. Dann werden die Actions verarbeitet und ein String zurückgegeben, der aussagt, welcher Navigationsfall als nächstes kommt. Listing 10 zeigt die *faces-config.xml* mit den Navigations-Rules. Dabei kann jeder *Command*-Komponente über das *actionRef*-Attribut eine Action zugewiesen werden. Dies ist der entscheidende Unterschied zu Struts, wo eine Action einen Request verarbeitet. Wenn es mehrere Buttons auf einer Seite gibt, muss zunächst ermittelt werden, welcher gedrückt wurde. Alternativ kann in der JSP mittels JavaScript die URL, die beim Submit des Formulars aufgerufen wird, abhängig vom geklickten Button, geändert werden. In JSF ist dies einfacher. Jede Action verarbeitet einen Klick auf einen Button, also genau eine bestimmte User-Action, die über das *actionRef*-Attribut an dem Button registriert wird.

Im Shop ist eine solche Action die Auswahl einer Artikelgruppe auf der Einstiegsseite (Abb. 2). Bei dieser Action wer-

den die Artikel der ausgewählten Gruppe vom Business Delegate gelesen, als aktuelle Artikel im Modell abgelegt und die Shop-Seite mit den Artikeln angezeigt. Listing 11 zeigt die entsprechende Action. JSF ruft zur Ausführung die *invoke*-Methode auf.

In der JSP wird eine Drop-Down-Liste mit den Artikelgruppen definiert, sowie ein Button, welcher die Menu-Action registriert hat:

```
<h:selectone_menu id="currentArticleGroup" valueRef=
                  "shoppingbean.currentArticleGroup">
  <h:selectitems valueRef="shoppingbean.articleGroup" />
</h:selectone_menu>
<h:command_button id="go"
                  actionRef="actionmanager.menuAction" label="Go"
                  commandName="go" />
```

Dabei wird auf die Action wie oben beschrieben über den Action Manager verwiesen. Auf der Shop-Seite selbst kann der Anwender erneut über die Drop-Down-Liste eine Artikelgruppe auswählen (Abb. 3). In diesem Fall wird dieselbe Action ausgeführt und die Seite mit den entsprechenden Artikeln neu aufgebaut. Wir haben also eine Wiederverwendung einer Action auf verschiedenen Seiten. Dies macht erneut deutlich, dass nicht wie bei Struts ein Request verarbeitet wird, sondern eine Aktion des Users, die unabhängig von der Seite sein kann. Diesmal muss aber der Warenkorb aktualisiert werden, da der User evtl. die Mengenangaben der zuletzt angezeigten Artikel geändert hat.

Dieses Event haben wir bereits mit dem *RecalcActionListener* implementiert. Es muss also keine neue Action definiert werden (neue Artikelgruppe anzei-

Anzeige

Abb. 5: Eingabe der Kundendaten

## Listing 12

### validate-Methode des Custom-Validators

```
public void validate(FacesContext context,
                    UIComponent component) {
    boolean valid = false;

    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    if (!(component instanceof UIOutput)) {
        return;
    }

    if (formatPatternsList == null) {
        // no patterns to match
        component.setValid(true);
        return;
    }

    String value = (((UIOutput)component).getValue().
                    toString());
    //validate the value against the list of valid patterns.
    Iterator patternIt = formatPatternsList.iterator();
    while (patternIt.hasNext()) {
        valid = isFormatValid(((String)patternIt.next()),
                               value);

        if (valid) {
            break;
        }
    }
    if (valid) {
        component.setValid(true);
    } else {
        component.setValid(false);
        Message errMsg = Resources.
            getMessageResources().getMessage(context,
            FORMAT_INVALID_MESSAGE_ID,
            (new Object[] {formatPatterns}));
        context.addMessage(component, errMsg);
    }
}
```

gen plus update Warenkorb), sondern es können die vorhandene Menu-Action und der vorhandne Listener wieder verwendet werden. Es muss lediglich der Listener zusätzlich zur Action beim Button registriert werden:

```
<h:selectone_menu id="currentArticleGroup" valueRef=
    "shoppingbean.currentArticleGroup">
    <h:selectitems valueRef="shoppingbean.articleGroup"/>
</h:selectone_menu>
<h:command_button id="go" label="Go" actionRef=
    "actionmanager.menuAction" commandName="go">
    <f:action_listener type="shopping.action.
        RecalcActionListener" />
</h:command_button>
```

Bei der Anzeige des Warenkorbs werden ebenfalls die vorhandenen Komponenten verwendet (Abb. 4). Die *CheckOutAction* setzt im Modell als aktuell angezeigte Artikel diejenigen aus dem Warenkorb. Bei der Anzeige wird wieder unsere Komponente *UiltemCount* verwendet, mit deren Hilfe der User die Mengenangaben ändern kann. Auch hier kann der Warenkorb mit dem *RecalcActionListener* aktualisiert und mit der *MenuAction* zurück zum Shop navigiert werden.

## Eigene Converter und Validator

Auf der *customer.jsp* muss der Kunde seine Daten eingeben (Abb. 5). Hier werden Custom- und Standard-Validator und ein Custom-Converter verwendet. Der Custom-Validator und der Custom-Converter sind aus dem *cardemo-example* des JSF-Tutorials kopiert.

Die Eingabe des Nachnamens ist Pflicht, folglich wird der Standard-Validator *Validate\_Required* verwendet. Der Validator erkennt in der *Process Validations Phase*, ob kein Wert angegeben wurde und erzeugt einen entsprechenden Fehler, der durch das *output\_errors*-Tag angezeigt wird:

```
<h:input_text id="lname" valueRef=
    "shoppingbean.customer.lastName">
    <f:validate_required />
</h:input_text>
<h:output_errors for="lname"/>
```

Für die Postleitzahl und die Kreditkartennummer ist ein bestimmtes Format

vorgeschrieben. Es kann also erneut der Standard-Validator *Validate\_Required* verwendet werden. Zudem benötigen wir einen Custom-Validator, der gegen ein angegebene Format prüft.

Um diesen zu implementieren, muss ein Validator-Tag und eine Validator-Klasse, welches das Interface *javax.faces.validator.Validator* implementiert, entwickelt werden. Der Validator muss zudem in der *faces-config.xml* deklariert werden. In der Validator-Klasse muss die *validate*-Methode implementiert werden (Listing 12). Diese wird von JSF in der *Process Validation Phase* aufgerufen. Ihr wird als Parameter die Komponente, die validiert wird, mitgegeben. Unsere Komponente holt sich den Wert aus der Komponente und prüft ihn gegen das im Validator-Tag angegebene Format. Der Komponente wird über die *setValid()*-Methode mitgeteilt, ob der Wert korrekt ist. Im Fehlerfall wird ein entsprechender Fehler im *FacesContext* abgelegt. Die Validatoren werden wie gehabt bei der Komponente in der JSP registriert:

```
<h:input_number id="zip" valueRef=
    "shoppingbean.customer.zip">
    <f:validate_required />
    <s:format_validator formatPatterns="99999"/>
</h:input_number>
<h:output_errors for="zip"/>
```

Neben dem Validator gibt es im Shop einen Custom-Converter für die Kreditkartennummer. Dieser konvertiert die eingegebenen Daten in ein modellkonformes, bzw. die Modell-Daten in ein für die anzeigeoptimiertes Format. Um einen eigenen Converter zu schreiben, muss das Interface *javax.faces.convert.Converter* implementiert und der Converter in der *faces-config.xml* deklariert werden.

Dabei gibt es zwei Methoden: *getAsObject(FacesContext, UIComponent, String)* wandelt in das Modell-Format. In unserem Converter werden alle Blanks und Bindestriche entfernt. Die Methode *getString(FacesContext, UIComponent, Object)* wandelt in das anzeigeoptimierte Format. Unser Converter fügt hierfür nach jeder vierten Stelle ein Blank zur besseren Lesbarkeit ein. Um den Converter zu verwenden, muss er als At-

tribut der Komponente bekannt gemacht werden:

```
<h:input_text id="ccno" valueRef="shoppingbean.  
    customer.creditcardNo" converter="creditcard">  
    <f:validate_required />  
    <s:format_validator formatPatterns=  
        "9999999999999999|9999 9999 9999|9999-9999-  
        9999-9999" />  
</h:input_text>  
<h:output_errors for="ccno" />
```

## Schlussbemerkung

Mit den JavaServer Faces kommt ein Framework, welches (aus der in Struts implementierten Request-Verarbeitung) einen Wandel zur Aktionsverarbeitung auf Basis von Dialogkomponenten und Listener vollzieht. Kenner der Swing-Konzepte werden sich schnell heimisch fühlen. Mit dem JSF-Lifecycle ist eine sauber abgegrenzte Verarbeitung der einzelnen Schritte vom Auslösen einer Aktion bis zum update des Modells und dem erneuten Rendern einer Response umgesetzt.

Allerdings schützt das Framework nicht vor unschönen Implementierungen. Ein sauberes Design ist nach wie vor nötig. In der *cardemo* des eingesetzten EA3-Releases gibt es z.B. einen einzigen großen ActionListener, der jede Aktion entgegennimmt und in einer großen *if-else*-Schleife prüft, welcher Button geklickt wurde und welche Aktion ausgeführt werden muss. Mit dem hier vorgestellten Design, mit der Trennung von Daten im Modell und der Verwaltung der Actions im Action Manager, kann jedoch eine sauber strukturierte Anwendung gebaut werden.

Wir möchten auch darauf hinweisen, dass das eingesetzte EA3-Release nur auf einem Tomcat 4.1 lief. Unter Tomcat 4.0 streikten die Samples. Ebenso hatten wir Probleme bei der Initialisierung der Beans in der *faces-config.xml*. Beim Start der Anwendung sollten alle Artikelgruppen vom Business Delegate geladen werden. Die implementierte *setInit(String)*-Methode im Action Manager, die wir

über das *managed-property*-Element aufrufen wollten, führte zu einem Fehler, da der Business Delegate noch nicht instanziiert war. Die einzelnen Beans werden wohl nicht beim Start, sondern beim ersten Aufruf instanziiert. Das Problem wurde mit einer *getInit()*-Methode gelöst, die ich von der Einstiegs-JSP aus aufrufe.

*Dipl. Ing. (M.E) Muhammet Öztürk und Dipl. Betriebswirt (Wirtschaftsinformatik, BA) Marco Michel sind Projektleiter bzw. IT-Architekten bei der Wüstenrot & Württembergische Informatik GmbH und haben mehrere Jahre Erfahrung als Softwareentwickler und Projektleiter in verschiedenen Softwareprojekten.*

## Links & Literatur

- Öztürk, Muhammet: „GUI fürs Web – Einführung in die Konzepte von JavaServer Faces (JSF)“, *Java Magazin* 1.2004
- JavaServer Faces (JSF): [java.sun.com/j2ee/javaserverfaces/](http://java.sun.com/j2ee/javaserverfaces/)
- Sun J2EE Patterns: [java.sun.com/blueprints/patterns/](http://java.sun.com/blueprints/patterns/)

# Anzeige