JavaServer Faces 2.0
Kapitel 4: JavaServer Faces im Detail

Max Mustermann

Uni Musterhausen
Fakultät Informatik

Sommersemester 2011

Grundlage der Folien

Bernd Müller
JavaServer Faces 2.0
Ein Arbeitsbuch für die Praxis
Hanser-Verlag
ISBN 978-3-446-41992-6
39,90 Euro
Amazon-Link

Ziel

Detailierte Beschreibung aller Konzepte, Mechanismen und Hintergründe zentraler JSF-Gebiete

- Bearbeitungsmodell
- JSF-Expression-Language
- Managed Beans
- Validierung und Konvertierung
- Event-Verarbeitung
- Navigation
- Internationalisierung
- Konfiguration
- Client-Ids und Komponenten-Ids
- Ressourcen-Behandlung

Bearbeitungsmodell einer JSF-Anfrage

HTTP, Servlets, MCV, Zustand, ...

Problem:

- JSF werden durch (ein) Servlet(s) implementiert
- damit HTTP-basiert
- wie passt das mit zustandsbehafteten Komponenten und MVC zusammen?

Lösung:

- *sehr* viel Aufwand treiben, um HTTP und dessen Nachteile zu verstecken:
  - Zustände der Komponenten implementieren
  - Event-Modell definieren (das alte, bekannte!)
  - ...daher ...
  - Aufteilen der Request-Bearbeitung in einzelne Schritte

Prinzipiell unterscheiden ob von/nach JSF-Seite

- Es gibt vier Kombinationsmöglichkeiten von JSF-Anfrage und JSF-Antwort mit anderen Anfragen und Antworten

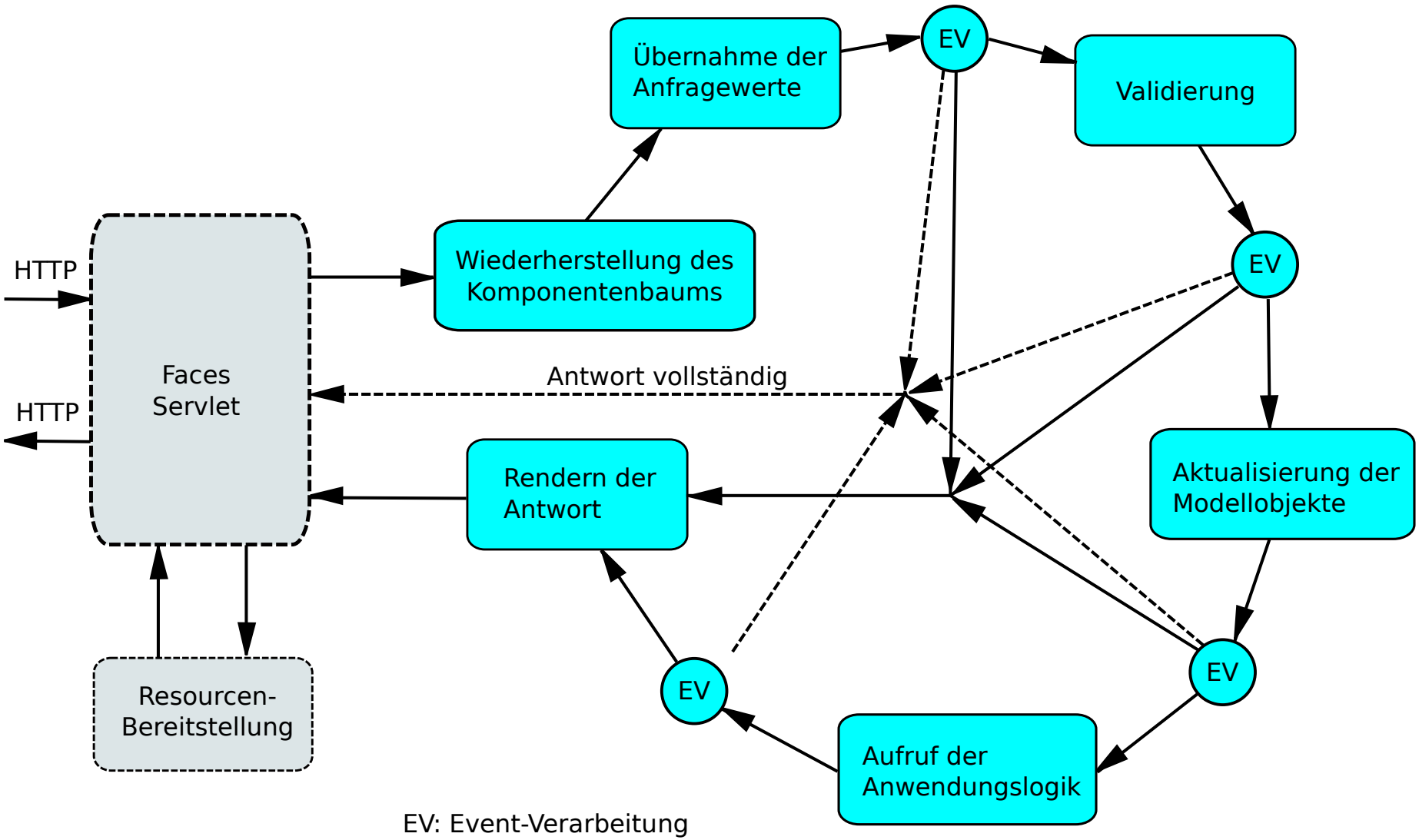
Tabelle 4.1: Möglichkeiten von Anfragen und Antworten

	JSF-Anfrage	andere Anfrage
JSF-Antwort	1	2
andere Antwort	3	4

- 1: JSF-Seite ruft JSF-Seite
- 2: z.B. HTML-Seite linkt auf JSF-Seite
- 3: z.B. JSF-Seite erzeugt PDF
- 4: hat nichts mit JSF zu tun

Die folgenden Ausführungen beziehen sich in der Regel auf den 1. Fall

Das Bearbeitungsmodell einer JSF-Anfrage



EV: Event-Verarbeitung

Wiederherstellung des Komponentenbaumes

Komponentenbaum

- die *View* ist eine nicht sichtbare Komponente und Wurzel des Komponentenbaumes einer Seite
- sie enthält damit alle Komponenten der Seite
- Komponenten werden zwischen der Antwort und einer erneuten Anfrage gespeichert
- dies kann auf dem Client oder dem Server geschehen
- der View wird eine View-Id zugeordnet, die aus dem Anfrage-URI besteht
- bei `/bank/pages/login.jsf` also `/pages/login.jsf`
- View-Id wird in Session gespeichert, daher feststellbar ob
  - Anfrage von JSF-Seite kommt (Alternative 1 in Tabelle 4.1), oder
  - zum ersten Mal besucht wird (Alternative 2 in Tabelle 4.1)

Speichern und Wiederherstellen

- in Alternative 1 wird gespeicherter Komponentenbaum wiederhergestellt
- im Alternative 2 wird ein neuer Komponentenbaum erstellt
- Komponentenbaum (neuer oder wiederhergestellter) wird dann im `FacesContext` gespeichert
- `FacesContext` enthält alle Informationen im Zusammenhang der Bearbeitung *einer* JSF-Anfrage
- Wiederherstellung umfasst auch Wiederherstellen aller verbundenen Event-Listener, Validierer, Konvertierer und Managed Beans
- wird Seite zum ersten Mal besucht (Alternative 2 Tabelle 4.1) wird zur Render-Response-Phase gesprungen, ansonsten zur Übernahme der Anfragewerte

Übernahme der Anfragewerte

Übernahme der Anfragewerte

- einige UI-Komponenten lassen Benutzereingaben zu
- das zugrunde liegende HTML-Formular schickt diese als POST-Request per HTTP an den Server
- dieser POST-String muss geparst werden und die Parameter mit ihren jeweiligen Werten müssen herausgefiltert werden
- diese Werte werden dann vorläufig den Komponenten zugewiesen
- vorläufig, weil Konvertierung und Validierung noch schief gehen können

Validierung

Konvertierung und Validierung

- jetzt sind alle Anfrageparameter für UI-Komponenten verfügbar
- vor Validierung muss noch konvertiert werden
- alle Post-Parameter sind Strings, daher in entsprechenden Typ konvertieren (automatisch für `Character`, `Boolean`, `Byte`, `Integer`, `Short`, `Long`, `Float`, `Double` und deren primitive Versionen sowie `BigDecimal`, `BigInteger`) und Enums
- Validierer überprüfen dann Eingaben
- eingebaut: `required="true"`
- eingebaut: Zahlen von bis, String-Länge, reguläre Ausdrücke
- jetzt der Komponente Wert zuweisen
- falls Wertänderung, `ValueChangeEvent` werfen

Aktualisierung der Modellobjekte

Aktualisierung der Modellobjekte

- jetzt sicher, dass alle Werte vom richtigen Typ und valide sind
- also den Modellobjekten zuweisen (bis jetzt alles ohne Modellobjekte)

Beispiel

```
<h:inputText id="betrag" required="true"
   value="#{ueberweisungHandler.betrag}">
```

Das Property `betrag` des Objekts `ueberweisungsHandler` bekommt den Wert des Eingabefeldes mit der Id `betrag` zugewiesen (Setter-Aufruf).

Aufruf der Anwendungslogik

Aufruf der Anwendungslogik

- bis jetzt alles automatisch ohne Anwendungslogik:
  - Decodieren
  - Konvertieren
  - Validieren
  - Wertzuweisung (Setter-Aufruf)
- Anwendungslogik durch Action-Listener

Action-Listener

Verschiedene Möglichkeiten (Details später)

```
<h:commandButton id="feld-0"
                 image="#{tttHandler.image[0]}"
    actionListener="#{tttHandler.zug}" />

<h:commandButton value="Anmelden"
    action="#{kundenHandler.login}" />

<h:commandButton id="button" value="OK"
                 action="success">
    <f:actionListener ...
```

Rendern der Antwort

Rendern der Antwort

- Rendern durch Spezifikation *nicht* festgelegt
- Jede JSF-konforme Implementierung muss mindestens JSP/HTML unterstützen
- Und seit 2.0 Facelets/XHTML
- zuletzt Abspeichern des Komponentenbaumes

Aufgabe

Laden Sie die Datei `WebContent/WEB-INF/web.xml` einer JSF-Anwendung in Ihren Editor. Der Context-Parameter `javax.faces.STATE_SAVING_METHOD` kann die Werte `client` und `server` haben. Testen Sie beide, indem Sie neu deployen und die erzeugten HTML-Seiten analysieren. Erkennen Sie den Unterschied?

Aufgabe

In Abschnitt 4.1.2 „Übernahme der Anfragewerte“ wurde der HTTP-POST-Request angesprochen. Verifizieren Sie diesen POST-Request bei einer beliebigen Anwendung, z.B. dem Online-Banking. Installieren Sie dazu das Firefox-Plugin Firebug.

JSF-Expression-Language

Bean-Properties

JSF-Expression-Language

- Die JSF-Expression-Language (JSF-EL) ähnelt der JSP-Expression-Language (syntaktisch)
- sie sind jedoch nicht kompatibel
- JSF-EL wird zwei Mal ausgewertet (Aktualisierung Modellobjekte und Render-Phase)
- Seit JSF 1.2 wird die *Unified Expression Language* verwendet
- Navigation über Objekt-Properties analog zu XPath
- Syntax: "#{ ... }"
  - String, Hash, in geschweifte Klammern eingeschlossen
- Inhalt: arith. oder log. Ausdrücke, Wertebindungen, Methodenbindungen
- Mit JSF 2.0 Verwendung aktuellster EL mit Methodenparametern

Bean-Properties

Bean-Properties

- eine *Wertebindung* (engl. Value Binding) bindet Wert einer UI-Komponente an Bean-Property (und umgekehrt)
- oder UI-Komponente selbst an Property
- oder UI-Komponenten-Property initialisiert
- eine *Methodenbindung* (engl. Method Binding) bindet Wert einer UI-Komponente an Bean-Methode (wird bei Event-Handlern und Validierungsmethoden verwendet)

Beispiele für Wertebindungen

```
<h:outputText value="Zugriff auf Bean-Properties:"
              style="font-weight: bold;" />
<h:outputText value="#{elHandler.name}" />
<h:outputText value="#{elHandler['name']}" />
<h:outputText value="Dies sind tolle #{elHandler.name}"
<h:outputText value="#{elHandler.array[0]}" />
<h:outputText value="#{elHandler.map['zwei']}" />
<h:outputText
    value="#{elHandler.map[elHandler.array[2]]}" />
```

Zeitpunkt der Auswertung

- Lesen (Getter) in der Phase *Rendern der Antwort*

Was ist elHandler?

Eine Managed-Bean der Klasse ElHandler

```
public class ELHandler {

    private String name = "Uebungen mit der Expression-L
    private Integer jahr = 2006;
    private String[] array = new String[]{ "eins", "zwei
    private Map map = new HashMap();

    public ELHandler() {
        map.put("eins", "Erster Map-Eintrag");
        map.put("zwei", "Zweiter Map-Eintrag");
        map.put("drei", "Dritter Map-Eintrag");
    }

}
```

Vordefinierte Variablen

Vordefinierte Variablen

Variablenname	Beschreibung
header	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist *ein* String.
headerValues	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist ein Array von Strings.
cookie	Eine `Map` von Cookies (`javax.servlet.http.Cookie`). Schlüssel ist der Cookie-Name.
initParam	Eine `Map` von Initialisierungsparametern der Anwendung. Diese werden im Deployment-Deskriptor definiert.

Vordefinierte Variablen (cont'd)

Variablenname	Beschreibung
param	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist *ein* String.
paramValues	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist ein Array von Strings.
facesContext	Die `FacesContext`-Instanz der aktuellen Anfrage.
component	Im Augenblick bearbeitete Komponente.
cc	Im Augenblick bearbeitete zusammengesetzte Komponente.
resource	Map von Ressourcen.
flash	Map von temporären Objekten für nächste View.

Vordefinierte Variablen (cont'd)

Variablenname	Beschreibung
view	Die aktuelle View.
viewScope	Eine `Map` von Variablen mit View-Scope.
request	Das Request-Objekt.
requestScope	Eine `Map` von Variablen mit Request-Scope.
session	Das Session-Objekt.
sessionScope	Eine `Map` von Variablen mit Session-Scope.
application	Das Application-Objekt.
applicationScope	Eine `Map` von Variablen mit Application-Scope.

Beispiele mit vordefinierten Variablen

```
<h:outputText
  value="Zugriff auf implizite JSF-EL-Objekte:"
  style="font-weight: bold;" />

<h:outputText value="#{header['User-Agent']}" />

<h:outputText value="#{view.locale}" />

<h:outputText
  value="#{initParam['javax.faces.STATE_SAVING_METHOD']}
```

Vergleiche, arithmetische und logische Ausdrücke

Arithmetische und logische Ausdrücke

- JSF-EL umfasst vollständige Arithmetik und Logik
- damit in der Regel kein Rückgriff auf Java nötig

Op	Alt.	Beschreibung
`.`		Zugriff auf eine Property, Methode oder einen Map-Eintrag
`[]`		Zugriff auf ein Array- oder Listen-Element oder einen Map-Eintrag
`()`		Klammerung für Teilausdrücke
`?:`		Bedingter Ausdruck: `<expr> ? <true-value> : <false-value>`
`+`		Addition
`-`		Subtraktion oder negative Zahl
`*`		Multiplikation
`/`	`div`	Division
`%`	`mod`	Modulo
`==`	`eq`	gleich (`equals()`-Methode)
`!=`	`ne`	ungleich
`<`	`lt`	kleiner
`>`	`gt`	größer
`<=`	`le`	kleiner-gleich
`>=`	`ge`	größer-gleich
`&&`	`and`	logisches UND
`||`	`or`	logisches ODER
`!`	`not`	logische Negation
`empty`		Test auf `null`, einen leeren String, oder Test auf Array, `Map` oder `Collection` ohne Elemente

Beispiele arithische und logische Ausdrücke

```
<h:outputText value="Arithmetische Ausdruecke:"
              style="font-weight: bold;" />
<h:outputText value="#{17 + 4}" />
<h:outputText
  value="Das uebernaechste Jahr ist #{elHandler.jahr + 2}" />
<h:outputText
  value="#{elHandler.jahr} ist #{((elHandler.jahr % 4)
    == 0 ? 'ein' : 'kein')} Schaltjahr" />

<h:outputText value="Boole'sche Ausdruecke:"
              style="font-weight: bold;" />
<h:outputText value="#{'konstante' == 'konstante'}" />
<h:outputText value="#{elHandler.jahr > 2000}" />
<h:outputText
  value="#{elHandler.jahr > 2000 && (elHandler.jahr %4 != 0)}"
```

Aufgabe

Erstellen Sie eine JSF-Seite, die mindestens fünf Parameter des HTTP-Request-Headers darstellt. Zuerst müssen Sie also mögliche Parameter recherchieren, sinnvollerweise in der HTTP-Spezifikation, der RFC 2616, und dort im Kapitel 14.

Aufgabe

Schreiben Sie einen einfachen Taschenrechner für die vier Grundrechenarten und z.B. ganze Zahlen. Die Operatoren können Sie explizit hart codieren.

Methodenaufrufe und Methodenparameter

Unified Expression-Language

- Unified EL Teil von JSP 2.1
- Zweites Maintenance-Release erweitert EL um Methoden mit Parametern
- JSF 2.0 nur JSP 2.1, nicht zweites MR
- GlassFish und JBoss-AS haben es aber

Beispiel

```
public String methodWithOneParam(String param) {
  return param + " " + param;
}

public String methodWithTwoParams(String param1,
                                  int param2) {
  return param1 + " " + param2;
}

public List<Integer> getList() { ... }
```

Beispiel

```
<h:outputText
    value="#{elHandler.methodWithOneParam('text')}"/>
<h:outputText
    value=
     "#{elHandler.methodWithTwoParams('text', 127)}"/>
<h:outputText value="#{elHandler.list.size()}" />
```

- Vor JSF 2.0 `size()` nicht möglich, da nicht Getter/Setter-Konvention
- Ersten beiden Methoden sowieso nicht möglich vor 2.0

Verwendung der Expression-Language in Java

Verwendung der EL in Java manchmal nötig/wünschenswert

```
public Integer getTestAusdruck() {
  FacesContext faces =
      FacesContext.getCurrentInstance();
  Application app = faces.getApplication();
  ExpressionFactory expressionFactory =
      app.getExpressionFactory();
  ELContext el = faces.getELContext();
  Integer val = (Integer) expressionFactory
        .createValueExpression(el, "#{17 + 4}",
                               Integer.class)
        .getValue(el);
  return val;
}
```

- Beispiel sinnlos, aber z.B. Zugriff auf vordefinierte Variablen möglich

Managed-Beans

Managed-Beans

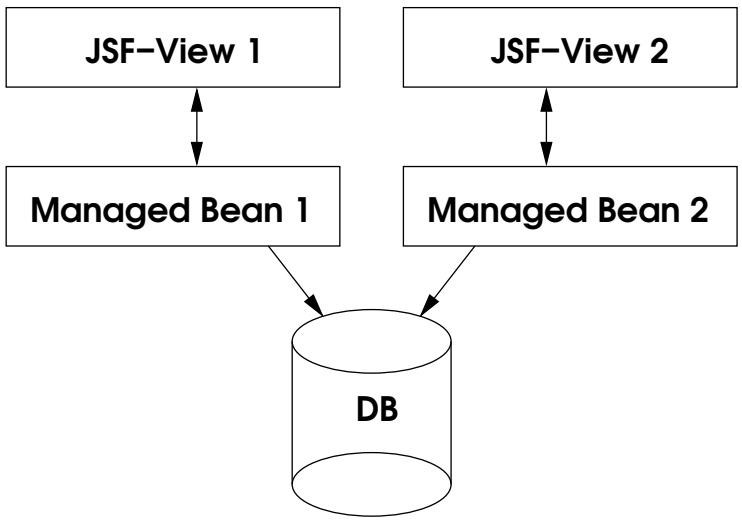
- Sammeln Daten von UI-Komponenten
- Implementieren Event-Listener
- Können Referenzen auf UI-Komponenten halten
- Bereits im Online-Banking verwendet: `KundenHandler`, `UeberweisungHandler`, ...
- Wir verwenden Managed-Bean und Handler synonym
- Managed-Beans können von JSF verwaltet werden
- Dazu `<managed-bean>`-Element in Konfig-Datei (in 4.3.5 Annotationen)
- Daher auch *Managed-Bean* genannt

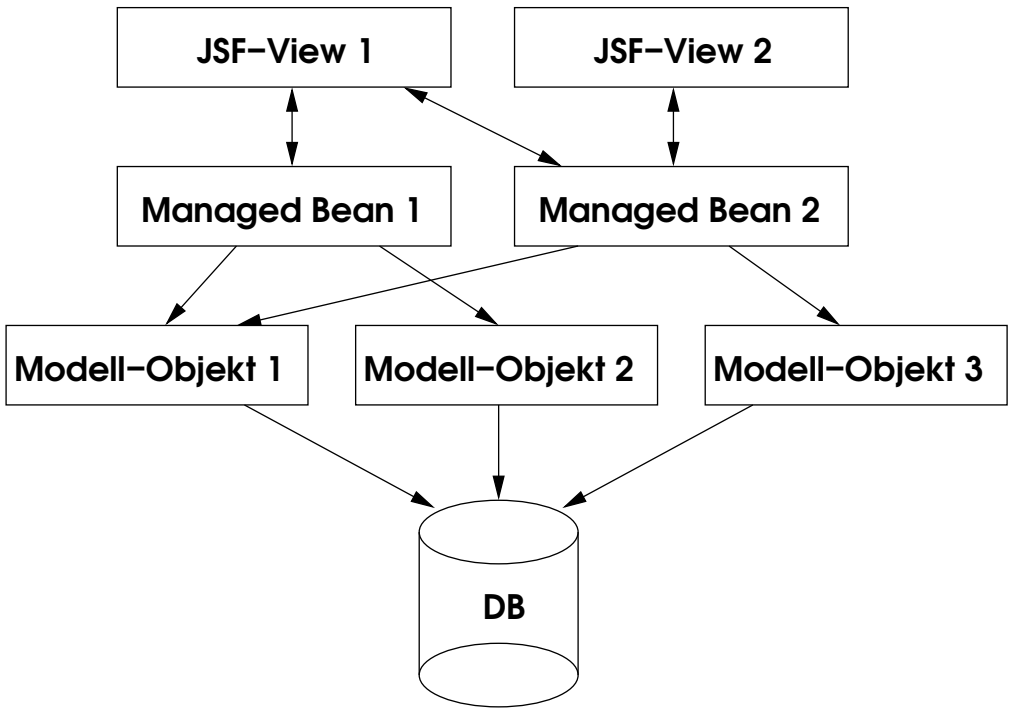
Architekturfragen

Einordnung Managed-Beans

- Managed-Beans gehören zum Controller (im MVC-Pattern)
- Zuordnung welche View/Model entscheidet Entwickler
- Möglich:
  - Managed-Bean zu genau einer View und direkter Datenzugriff
  - oder über „richtiges“ Modell, also Geschäftsobjekte

Einordnung Managed-Beans





Automatische Verwaltung von Managed-Beans

Managed-Beans

- werden vom JSF-Laufzeitsystem automatisch verwaltet
- Konfiguration in Datei `faces-config.xml` (Default)
- damit:
  - Beans an *einer* Stelle deklariert
  - Beans an *einer* Stelle initialisiert
  - Lebensdauer deklarativ beschrieben
  - Konfiguration nicht programmatisch
  - Verwendung von Wertebindungen möglich

Beispiel Managed-Bean

```
<managed-bean>
  <description>Einfache Managed-Bean</description>
  <managed-bean-name>mbHandler</managed-bean-name>
  <managed-bean-class>
    de.jsfpraxis.mb.MBHandler
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <description>Der Titel der Seite</description>
    <property-name>title</property-name>
    <value>Beispiel zu Managed-Beans</value>
  </managed-property>
  <managed-property>
    <description>
      Eine Zahl zur Demonstration der automatischen Konvertieru
    </description>
    <property-name>pi</property-name>
    <value>3.1415</value>
  </managed-property>
</managed-bean>
```

Lebenszeit,Gültigkeitsbereich, Konvertierung

- Bean wird beim ersten Zugriff automatisch erzeugt
- dazu wird Default-Konstruktor aufgerufen
- Gültigkeitsbereich: (`<managed-bean-scope>`)

none	Initialisierung anderer Beans
request	für diesen (HTTP-) Request
view	für die View
session	für diese (HTTP-) Session
application	global für die Anwendung

- `title` ist String: kein Problem
- `pi` ist `Double`: ebenfalls kein Problem, wird automatisch gemacht

Initialisierung

Initialisierung

- schon gesehen, kein Problem für einfache Typen
- bei Arrays, Listen und Maps etwas komplizierter
- Arrays, Listen:
  in `<list-entries>` eingeschlossene `<value>`
- Maps:
  in `<map-entries>` eingeschlossene `<map-entry>` mit jeweils `<key>` und `<value>`

Beispiel Array/List

faces-config.xml:

```
<managed-bean>
  <managed-bean-name> ...
  <managed-bean-class> ...
  <managed-bean-scope> ...
  <managed-property>
    <description>Wichtige URLs</description>
    <property-name>urls</property-name>
    <list-entries>
     <value>http://java.sun.com/docs/books/jls/index.html</valu
     <value>http://java.sun.com/j2ee/javaserverfaces</value>
     <value>http://java.sun.com/j2se/1.5.0/download.jsp</value>
     <value>http://www.jsfpraxis.de</value>
    </list-entries>
  </managed-property>
  ...
```

Java:

```
  private List urls = new ArrayList();
```

Beispiel Array/List in JSF-Seite

```
<h:dataTable var="url" value="#{mbHandler.urls}">
  <f:facet name="header">
    <h:outputText value="Wichtige URLs" />
  </f:facet>
  <h:column>
    <h:outputText value="#{url}" />
  </h:column>
</h:dataTable>
```

Komponentenbindung

Komponentenbindung

anstatt `<h:outputText value="Wichtige URLs"/>`

mit Komponentenbindung:

```
<h:outputText value="Wichtige URLs"
              binding="#{mbHandler.ueberschrift}"/>
```

Property:

```
private javax.faces.component.html.HtmlOutputText ueberschrift
```

Verwendung:

```
ueberschrift.setValue("Ganz wichtige URLs");
```

Jetzt neuer Text in der Seite!!!

Java-EE-5-Annotationen

Dependency-Injection

- Resource-Injection mit Java-EE-5 eingeführt
- Spezielle Form der Dependency-Injection
- Ressourcen z.B.
  - Persistenzkontexte
  - JDBC-Datenquellen
  - JMS-Nachrichtenziele

JSR 250 Common Annotations (nicht EE 5)

Annotation	**Beschreibung**
Package `javax.annotation`	
`@PostConstruct`	Annotierte Methode wird aufgerufen, nachdem alle Injektionen erfolgt sind.
`@PreDestroy`	Annotierte Methode wird aufgerufen, bevor Objekt gelöscht wird.
`@Resource`	Injiziert allgemeine Ressource.
`@Resources`	Fasst mehrere `@Resource`-Annotationen zusammen.

Enterprise JavaBeans

Annotation	**Beschreibung**
Package `javax.ejb`	
`@EJB`	Injiziert Session-Bean.
`@EJBs`	Fasst mehrere `@EJB`-Annotationen zusammen.

Java Persistence API

Annotation	**Beschreibung**
Package `javax.persistence`	
`@PersistenceContext`	Injiziert Entity-Manager als Persistenzkontext.
`@PersistenceContexts`	Fasst mehrere `@PersistenceContext`-Annotationen zusammen.
`@PersistenceUnit`	Injiziert Entity-Manager-Fabrik als Persistenzeinheit.
`@PersistenceUnits`	Fasst mehrere `@PersistenceUnit`-Annotationen zusammen.

Web-Services

Annotation	**Beschreibung**
Package `javax.xml.ws`	
`@WebServiceRef`	Injiziert Referenz auf Web-Service.
`@WebServiceRefs`	Fasst mehrere `@WebServiceRef`-Annotationen zusammen.

Annotationen

Annotationen

- Hielten Einzug in JSF 2.0
- Alternative zu `faces-config.xml`
- Bei Widersprüchen gilt XML
- Damit konfigurierbare Systeme ohne Code-Änderung/Compilation
- Hier jetzt nur Annotationen für Managed-Beans

Annotationen für Managed-Beans

- Prinzipiell das, was `faces-config.xml` auch möglich:
  - Dass überhaupt Managed-Bean: `@ManagedBean`
  - Name: Default und `name="..."`
  - Scope: `@RequestScoped`, `@SessionScoped`, `@ApplicationScoped`, `@NoneScoped`, `@ViewScoped`
  - Managed Properties: `@ManagedProperty`

Beispiel

```
@ManagedBean(name = "mbAnnotationHandler")
@SessionScoped
public class MBAnnotationHandler {

   @ManagedProperty(value = "Annotierte Managed Beans")
   private String title;

   @ManagedProperty(value = "3.1415")
   private Double pi;

   ...
}
```

- Siehe API-Doc

Validierung und Konvertierung

Validierung

- syntaktische Validierung
  - Eingabe vorhanden / nicht vorhanden
  - Eingabe hat Format xx.xx.xxxx
  - Eingabe hat Format (1..31).(1..12).(1990..2090)
  - ...
- semantische Validierung
  - Eingabe ist Datum
  - Eingabe ist Datum und liegt nach anderem Datum
  - Benutzername und Passwort passen zusammen
  - ...

Konvertierung

- von String nach Integer, BigDecimal, ... und umgekehrt

Standardkonvertierer

Allgemeines

- jeder Konvertierer muss Interface `javax.faces.convert.Converter` implementieren
- mit den Methoden

```
public java.lang.Object getAsObject(
    javax.faces.context.FacesContext context,
    javax.faces.component.UIComponent component,
    java.lang.String value)

public java.lang.String getAsString(
    javax.faces.context.FacesContext context,
    javax.faces.component.UIComponent component,
    java.lang.Object value)
```

Die Standardkonvertierer

- Standardkonvertierer ebenfalls im Package `javax.faces.convert`
- Character, Boolean, Byte, Integer, Short, Long, Float, Double
- char, boolean, byte, int, short, long, float, double
- BigDecimal, BigInteger
- Aufzählungstypen (Enums), siehe Abschnitt 4.4.3
- diese Konvertierer immer automatisch, wenn Wertebindung mit Property von entsprechendem Typ

Konvertierung ganzer Zahlen (JSF)

```
<h:panelGrid columns="2">
  <h:outputLabel for="byteValue" value="Byte-Wert:" />
  <h:inputText id="byteValue" size="30"
     value="#{ganzeZahlenHandler.byteValue}" />
  <h:outputLabel for="shortValue" value="Short-Wert:" />
  <h:inputText id="shortValue" size="30"
     value="#{ganzeZahlenHandler.shortValue}" />
  <h:outputLabel for="intValue" value="Int-Wert:" />
  <h:inputText id="intValue" size="30"
     value="#{ganzeZahlenHandler.intValue}" />
  <h:outputLabel for="longValue" value="Long-Wert:" />
  <h:inputText id="longValue" size="30"
     value="#{ganzeZahlenHandler.longValue}" />
  <h:outputLabel for="bigIntValue" value="BigInteger-Wer
  <h:inputText id="bigIntValue" size="30"
     value="#{ganzeZahlenHandler.bigIntValue}" />
</h:panelGrid>
```

Konvertierung ganzer Zahlen (Java)

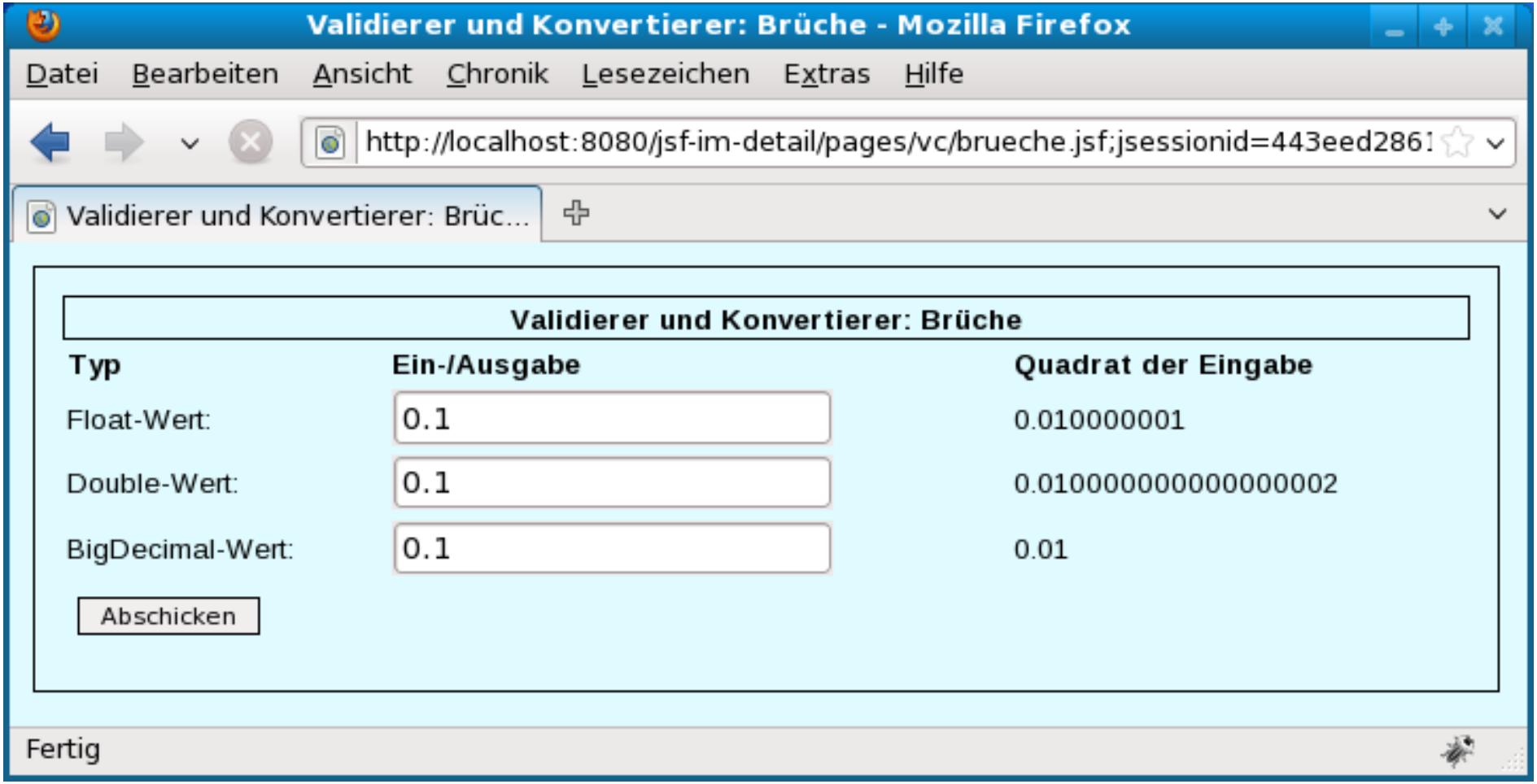
```
private Byte byteValue;
private Short shortValue;
private Integer intValue;
private Long longValue;
private BigInteger bigIntValue;
```

Konvertierung gebrochener Zahlen (JSF)

```
<h:panelGrid columns="3">
  <h:outputText value="Typ" style="font-weight: bold;" />
  <h:outputText value="Ein-/Ausgabe"
                style="font-weight: bold;" />
  <h:outputText value="Quadrat der Eingabe"
     style="font-weight: bold;" />

  <h:outputLabel for="floatValue" value="Float-Wert:" />
  <h:inputText id="floatValue"
     value="#{bruecheHandler.floatValue}" />
  <h:outputText value="#{bruecheHandler.floatValueQuadrat}" />

  <h:outputLabel for="doubleValue" value="Double-Wert:" />
  <h:inputText id="doubleValue"
     value="#{bruecheHandler.doubleValue}" />
  <h:outputText value="#{bruecheHandler.doubleValueQuadrat}" />

  <h:outputLabel for="bigDecimalValue"
                 value="BigDecimal-Wert:" />
  <h:inputText id="bigDecimalValue"
     value="#{bruecheHandler.bigDecimalValue}" />
  <h:outputText
```

Konvertierung gebrochener Zahlen (Java)

```
private Float floatValue;
private Double doubleValue;
private BigDecimal bigDecimalValue;

public Float getFloatValueQuadrat() {
  if (floatValue == null)
    return (float) 0.0;
  return floatValue * floatValue;
}

public Double getDoubleValueQuadrat() {
  if (doubleValue == null)
    return 0.0;
  return doubleValue * doubleValue;
}

public BigDecimal getBigDecimalValueQuadrat() {
  if (bigDecimalValue == null)
    return new BigDecimal(0);
  return bigDecimalValue.multiply(bigDecimalValue);
}
```



Merke

Rechne nie mit Float und Double, denn das gibt nur Trouble

Kalenderdaten und Zahlen

Neben der „reinen“ Konvertierung von Zahlen gibt es noch eine Reihe von unterschiedlichen Darstellungen, die auf regionalen, sprachlichen und kulturellen Unterschieden basieren.

JSF bietet ein sehr großes Spektrum, um Kalenderdaten und Zahlen unterschiedlich bzgl. dieser Eigenschaften darzustellen:

`<f:convertDateTime>`
`<f:convertNumber>`

Beispiel <f:convertDateTime>

```
<h:outputText value="#{dtHandler.date}">
  <f:convertDateTime type="date" />
</h:outputText>
<h:outputText value="#{dtHandler.date}">
 <f:convertDateTime type="time" />
</h:outputText>
<h:outputText value="#{dtHandler.date}">
  <f:convertDateTime type="both" />
</h:outputText>
```

wird zu

```
Jan 8, 2006
7:51:28 PM
Jan 8, 2006 7:51:28 PM
```

Property ist `java.util.Date`, Default-Lokalisierung ist Englisch

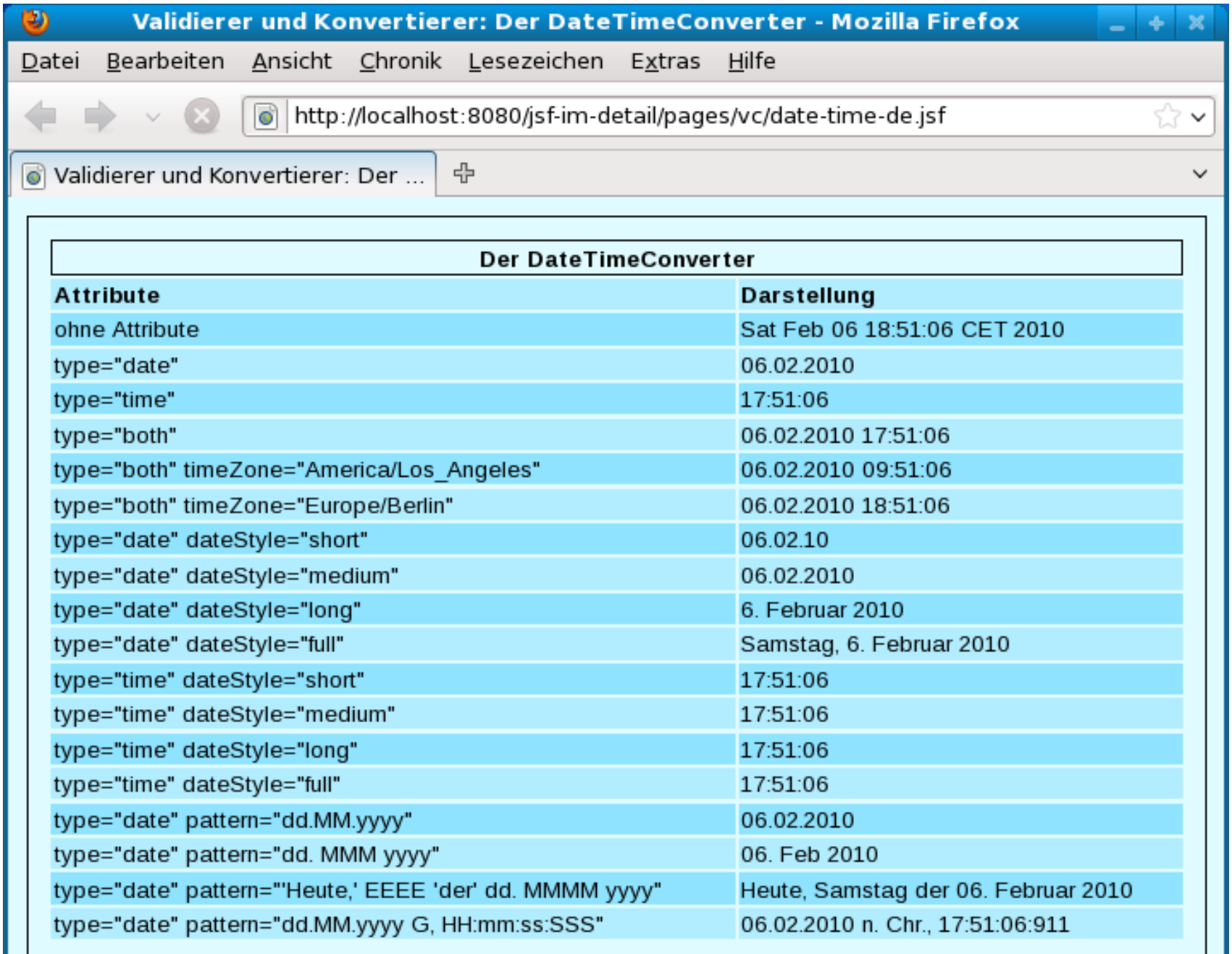
Tabelle 4.4: Attribute des <f:convertDateTime>

Attribut	Werte und Beschreibung
type	date (Default), time oder both. Anzeige von Datum, Zeit, oder Datum und Zeit.
dateStyle	short, medium (Default), long und full. Formatangabe für den Datumteil, falls type gesetzt.
timeStyle	short, medium (Default), long und full. Formatangabe für den Zeitteil, falls type gesetzt.
timeZone	Angabe der Zeitzone. Falls nicht gesetzt, ist der Default Greenwich-Mean-Time (GMT).
locale	Lokalisierung, entweder als Instanz von java.util.Locale oder als String.
pattern	Angabe eines Patterns zur Formatierung. Alternative zu type. Details siehe Tabelle 4.5.

Tabelle 4.5: Zeichen des pattern-Attributs

Zeichen	Bedeutung	Beispiel	Darstellung
G	Epoche	G	n. Chr.
y	Jahr	yyyy	2006
M	Monat des Jahres	MM MMM	06 Juni
w	Woche des Jahres	w	14
W	Woche im Monat	W	3
d	Tag im Monat	dd	05
D	Tag im Jahr	D	21
E	Tag in der Woche	EE EEEE	Mo Montag
h	Stunde (0–12, am/pm)	h	8
H	Stunde (0–23)	HH	08
m	Minuten	mm	26
s	Sekunden	ss	59
S	Millisekunden	SSS	123
’	Fluchtzeichen für Text	’Heute’	Heute
’’	Apostroph	’’	’

Beispiele zur Konfiguration des DateTimeConverters



Der DateTimeConverter	
Attribute	**Darstellung**
ohne Attribute	Sat Feb 06 18:51:06 CET 2010
type="date"	06.02.2010
type="time"	17:51:06
type="both"	06.02.2010 17:51:06
type="both" timeZone="America/Los_Angeles"	06.02.2010 09:51:06
type="both" timeZone="Europe/Berlin"	06.02.2010 18:51:06
type="date" dateStyle="short"	06.02.10
type="date" dateStyle="medium"	06.02.2010
type="date" dateStyle="long"	6. Februar 2010
type="date" dateStyle="full"	Samstag, 6. Februar 2010
type="time" dateStyle="short"	17:51:06
type="time" dateStyle="medium"	17:51:06
type="time" dateStyle="long"	17:51:06
type="time" dateStyle="full"	17:51:06
type="date" pattern="dd.MM.yyyy"	06.02.2010
type="date" pattern="dd. MMM yyyy"	06. Feb 2010
type="date" pattern="'Heute,' EEEE 'der' dd. MMMM yyyy"	Heute, Samstag der 06. Februar 2010
type="date" pattern="dd.MM.yyyy G, HH:mm:ss:SSS"	06.02.2010 n. Chr., 17:51:06:911

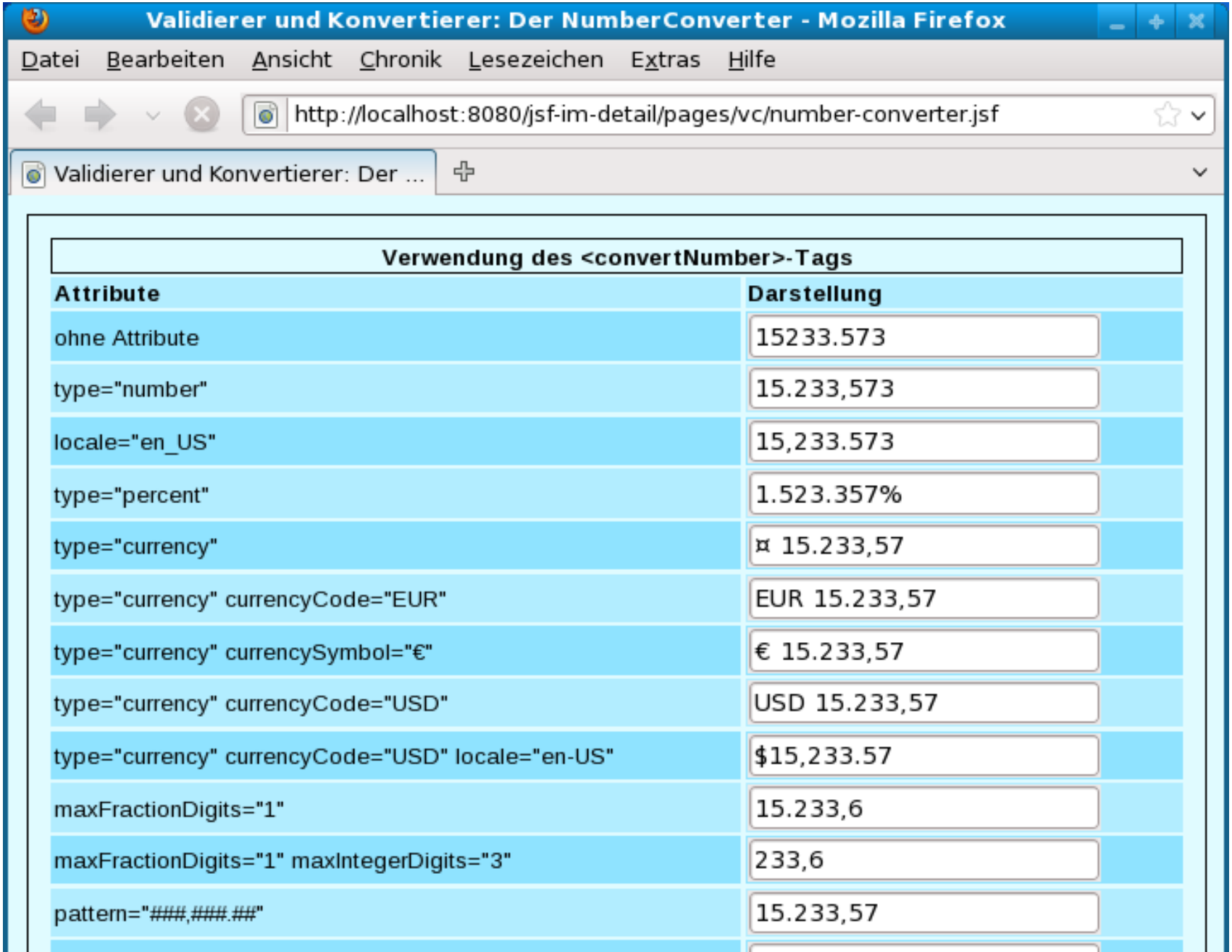
Tabelle 4.6: Attribute des <f:convertNumber>

ttribut	Werte und Beschreibung
ype	number (Default), currency oder percent. Anzeige einer Zahl, einer Währung oder eines Prozentsatzes.
ocale	Lokalisierung, entweder als Instanz von java.util.Locale oder als String.
urrencyCode	Dreistelliger Währungs-Code nach ISO 4217. Nur möglich, wenn type=currency. Alternative zu currencySymbol.
urrencySymbol	Währungssymbol, nur möglich, wenn type=currency. Alternative zu currencyCode.
inFractionDigits	Minimale Anzahl von Nachkommastellen.
axFractionDigits	Maximale Anzahl von Nachkommastellen.
inIntegerDigits	Minimale Stellenzahl der Ganzzahl.
axIntegerDigits	Maximale Stellenzahl der Ganzzahl.
roupingUsed	Schalter zur Anzeige des Gruppierungszeichens. Default ist true.
ntegerOnly	Schalter, ob nur ganzzahliger Anteil der Eingabe verarbeitet werden soll. Default ist false.
attern	Angabe eines Patterns zur Formatierung. Alternative zu type. Details siehe Tabelle 4.7.

Tabelle 4.7: Zeichen des `pattern`-Attributs

Zeichen	Bedeutung
0	Eine Ziffer.
#	Eine Ziffer. Führende Null, sofern notwendig.
.	Dezimalseparator der aktuellen Lokalisierung.
,	Gruppierungssymbol der aktuellen Lokalisierung.
'	Fluchtzeichen für Text.

Beispiele zur Konfiguration des `NumberConverters`



Konvertierung von Aufzählungstypen

Enum-Konvertierer

- Eingebauter Konvertierer für Enums, direkt verwendbar

```
public enum Familienstand {
  LEDIG, VERHEIRATET, GESCHIEDEN, VERWITWET
}

@ManagedBean
public class EnumHandler {

  private Familienstand familienstand;
  ...
```

Verwendung in JSF-Seite

- Sinnvollerweise in einem Auswahlmenü

```
<h:selectOneMenu id="enum"
        value="#{enumHandler.familienstand}">
  <f:selectItem itemLabel="ledig"
                itemValue="LEDIG" />
  <f:selectItem itemLabel="verheiratet"
                itemValue="VERHEIRATET" />
  <f:selectItem itemLabel="geschieden"
                itemValue="GESCHIEDEN" />
  <f:selectItem itemLabel="verwitwet"
                itemValue="VERWITWET" />
</h:selectOneMenu>
```

Bessere Alternative

```
<h:selectOneMenu
      value="#{enumHandler.familienstand}">
  <f:selectItems value="#{enumHandler.values}"/>
</h:selectOneMenu>

public Object[] getValues() {
  SelectItem[] items =
    new SelectItem[Familienstand.values().length];
  for (int i = 0; i < items.length; i++) {
    items[i] =
        new SelectItem(Familienstand.values()[i],
      Familienstand.values()[i]
        .toString().toLowerCase());
  }
  return items;
}
```

Anwendungsdefinierte Konvertierer

Use-Case

- Fachklasse `Kreditkartennummer` existiert
- Eingabe soll mit Leerzeichen oder „-“ erfolgen können

```
public class Kreditkartennummer {

  public Kreditkartennummer(String q1, String q2,
                            String q3, String q4) {
    ...
  }

  public String toString() { ... }
}
```

Konvertierer muss Interface Converter implementieren

```
public class KreditkartennummerConverter
        implements Converter {
  public Object getAsObject(FacesContext context,
              UIComponent component, String value) {
    Kreditkartennummer nummer = null;
    String[] quads = null;
    if (value.indexOf('-') != -1) {
      quads = value.trim().split("-");
    } else {
      quads = value.trim().split(" ");
    }
    if (quads.length != 4) {
      FacesMessage message =
      new FacesMessage(FacesMessage.SEVERITY_ERROR,
                       "Keine Kreditkartennummer",
                       "Keine Kreditkartennummer");
      throw new ConverterException(message);
    }
    ...
```

Konvertierer muss Interface Converter implementieren (cont'd)

```
    try {
      nummer = new Kreditkartennummer(quads[0], quads[1]
                                      quads[2], quads[3]
    } catch (Exception e) {
      FacesMessage message =
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
              "Keine Kreditkartennummer",
              "Keine Kreditkartennummer");
      throw new ConverterException(message);
    }
    return nummer;
  }

  public String getAsString(FacesContext context,
              UIComponent component, Object value) {
    return ((Kreditkartennummer) value).toString();
  }
}
```

Verwendung (global)

```
<h:inputText
 value="#{kreditkartenHandler.kreditkartennummer}"/>

@FacesConverter(forClass=Kreditkartennummer.class)
public class KreditkartennummerConverter
             implements Converter {
  ...
```

Alternativ in `faces-config.xml`

```
<converter>
  <converter-class>KreditkartennummerConverter</converte
  <converter-for-class>Kreditkartennummer</converter-for
</converter>
```

Verwendung (lokal)

```
<h:inputText
  value="#{kreditkartenHandler.kreditkartennummer}"
  converter="kreditkartenkonvertierer" />
```

Alternativ

```
<h:inputText value="#{kreditkartenHandler.kreditkartennu
  <f:converter converterId="kreditkartenkonvertierer"/>
</h:inputText>
```

Woher kommt der Name?

```
@FacesConverter("kreditkartenkonvertierer")
public class KreditkartennummerConverter
             implements Converter {
  ...
```

Alternativ in `faces-config.xml`

```
<converter>
  <converter-id>kreditkartenkonvertierer</converter-id
  <converter-class>KreditkartennummerConverter</conver
</converter>
```

Standardvalidierer

Standardvalidierer

- Validierung Hauptaufgabe eines GUIs
- JSF enthält einige Validierer
- und es ist sehr einfach, weitere hinzuzufügen
- es gibt die Validierer
  - `LengthValidator`
  - `LongRangeValidator`
  - `DoubleRangeValidator`
  - `RegexValidator` (2.0)
  - `RequiredValidator` (2.0)
- alle im Package `javax.faces.validator`
- überprüfen Länge, Bereich, regulären Ausdruck und Existenz

Standardvalidierer (cont'd)

- Verwendung einfach: in Eingabe enthaltenes
  - `<f:validateLength>`
  - `<f:validateLongRange>`
  - `<f:validateDoubleRange>`
- mit Attributen `minimum` und `maximum`

Beispiel

```
<h:inputText value="#{eingabeHandler.longValue}"
             required="true">
  <f:validateLongRange minimum="100" maximum="500" />
</h:inputText>
```

Standardvalidierer (cont'd)

- Verwendung
  - `<f:validateRegex>` mit reg. Ausdruck im Attribut `pattern`
  - `<f:validateRequired>`

Beispiel

```
<h:inputText id="email"
             value="#{eingabeHandler.email}">
  <f:validateRequired />
  <f:validateRegex pattern=".+@.+\..+" />
</h:inputText>
```

- `<f:validateRequired>` kann auch mehrer Eingaben *umfassen*

Standardvalidierer (cont'd)

- Bereichsvalidierer nicht auf einfache Texteingaben (`<h:inputText>`) beschränkt, sondern etwa auch in
  - Drop-down-Liste (`<h:selectOneMenu>`)
  - Check-Box (`<h:selectManyCheckbox>`)
  - Radio-Button (`<h:selectOneRadio>`)
  - ...
- dann aber „etwas konstruierter“, da bestimmte Wahlmöglichkeiten trotz Anzeige nicht valide sind

Drop-down-Liste mit Validator

```
<h:panelGrid columns="2" styleClass="borderTable" border="1">
  <h:outputText value="Waehlen Sie einen Wert zwischen
     #{eingabeHandler.min} und #{eingabeHandler.max}" />
  <h:selectOneMenu id="zahlenauswahl" required="true"
     value="#{eingabeHandler.menueauswahl}">
    <f:selectItem itemValue="" itemLabel="" />
    <f:selectItem itemValue="1" itemLabel="eins" />
    <f:selectItem itemValue="2" itemLabel="zwei" />
    <f:selectItem itemValue="3" itemLabel="drei" />
    <f:selectItem itemValue="4" itemLabel="vier" />
    <f:selectItem itemValue="5" itemLabel="fuenf" />
    <f:selectItem itemValue="6" itemLabel="sechs" />
    <f:selectItem itemValue="7" itemLabel="sieben" />
    <f:selectItem itemValue="8" itemLabel="acht" />
    <f:selectItem itemValue="9" itemLabel="neun" />
    <f:validateLongRange minimum="#{eingabeHandler.min}"
       maximum="#{eingabeHandler.max}" />
  </h:selectOneMenu>
  <h:commandButton action="#{eingabeHandler.abschicken}"
     value="Abschicken" />
  <h:outputText
     value="Auswaehlt wurde: #{eingabeHandler.menueauswahl}"/>
```

Validierungsmethoden

Eigene Valierer / Validierungsmethoden

- Standardvalidierer relativ eingeschränkt
- mögliche (Anwendungs-)Wünsche:
  - Überweisungsbetrag gedeckt ?
  - E-Mail-Adresse syntaktisch (semantisch?) korrekt ?
  - passt PLZ zu Ort ?
  - ...
- möglich mit:
  - Validierungsmethode
  - Validiererklasse

Validierungsmethoden

- jede Eingabekomponente kennt Attribut `validator`
- ist Methodenbindung für Validierungsmethode

Beispiel:

```
<h:inputText id="eingabe"
    validator="#{eingabeHandler.validateEmail}"
    value="#{eingabeHandler.textValue}"
    required="true" />
```

- Methode `validateEmail` wird an Property `textValue` gebunden

Validierungsmethoden II

Methode der Managed-Bean `eingabeHandler`

```
public void validateEmail(FacesContext context,
                          UIComponent component,
                          Object value)
  throws ValidatorException {
  if (!((String) value).matches(".+@.+\\..+")) {
    throw new ValidatorException(
      new FacesMessage("Fehlerhafte E-Mail-Syntax"));
  }
}
```

Parameter:

- Faces-Context des aktuellen Requests
- Komponente, deren Wert zu validieren ist
- der Wert selbst

Entwickeln von Validierungsmethoden

- Methode muss selbe Signatur wie Methode `validate()` des Interface `javax.faces.validator.Validator` haben, allerdings Methodenbezeichner beliebig
- muss `ValidatorException` bei negativem Validierungsergebnis werfen

Aufgabe

Schreiben Sie eine Validierungsmethode, die prüft, ob eine Texteingabe eine Kreditkartennummer ist. Kreditkartennummern sind 16 Ziffern, die in Vierergruppen angeordnet und durch ein Minus- oder Leerzeichen getrennt sind. Also z.B. „1234-5678-9876-5432“ oder „1234 5678 9876 5432“. Eine inhaltliche Überprüfung soll nicht vorgenommen werden. Verknüpfen Sie die Validierungsmethode mit einer JSF-Eingabe, und testen Sie die Seite.

Anwendungsdefinierte Validierer

Eigene Validierer

- Analog zu Konvertierern auch eigenen Validiererklassen möglich
- Können aber ausschließlich über Ihre Id verwendet werden
- Daher nur über Namen (Annotationen oder XML) verwendbar

Die Klasse EmailValidator

```
@FacesValidator("emailvalidierer")
public class EmailValidator implements Validator {

  public void validate(FacesContext context,
                       UIComponent component,
                       Object value)
          throws ValidatorException {
    if (!((String) value).matches(".+@.+\\..+")) {
      throw new ValidatorException(
          new FacesMessage(
                  FacesMessage.SEVERITY_ERROR,
                  "Fehlerhafte E-Mail-Syntax",
                  "Fehlerhafte E-Mail-Syntax"));
    }
  }
}
```

Verwendungsalternativen

```
<h:inputText id="eingabe"
     validator="emailvalidierer"
     value="#{eingabeHandler.textValue}"
     required="true" />
```

oder

```
<h:inputText id="eingabe"
     value="#{eingabeHandler.textValue}"
     required="true">
   <f:validator validatorId="emailvalidierer"/>
</h:inputText>
```

In faces-config.xml

```
<validator>
  <validator-id>emailvalidierer</validator-id>
  <validator-class>EmailValidator</validator-class>
</validator>
```

Eingabekomponenten und das immediate-Attribut

Das `immediate`-Attribut

- Eingabekomponenten besitzen `immediate`-Attribut
- Ist dieses `true` findet Validierung und Konvertierung schon bei der Übernahme der Anfragewerte und nicht in der Validierungsphase statt
- Warum / wo wird das benötigt?

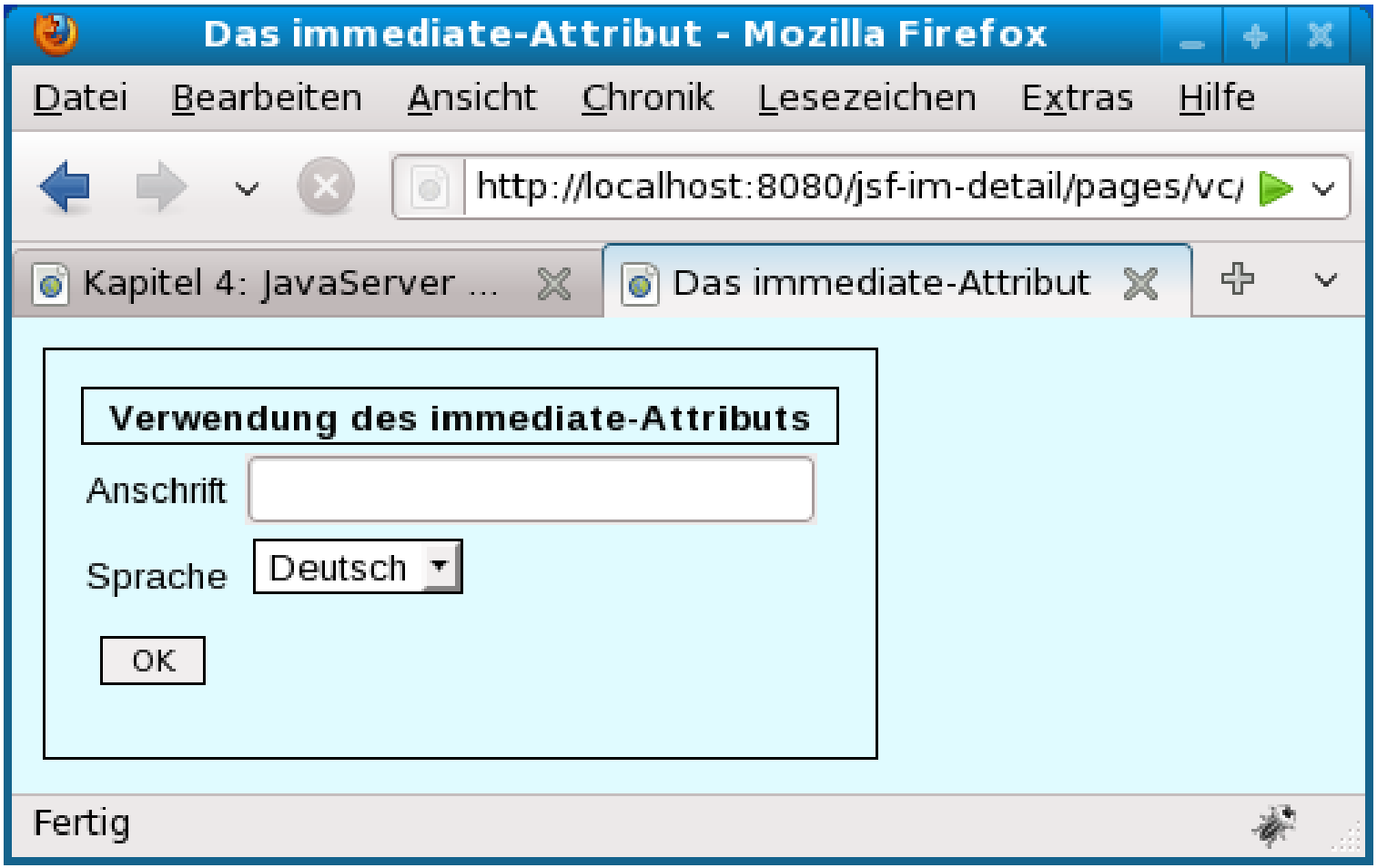
Anwendungsfall

- Auswahl der Sprache über Menü inklusive Änderung
- Abschicken des Formulars mit Anwendungslogik
- Aber nur, falle Eingaben ausgefüllt sind

Anwendungsfall: Code

```
<h:panelGrid columns="2">
  <h:outputLabel
      value="#{immediateHandler.addressLabels[
                    immediateHandler.language]}" />
  <h:inputText value="#{immediateHandler.address}"
      required="true" />
  <h:outputLabel
      value="#{immediateHandler.languageLabels[
                    immediateHandler.language]}" />
  <h:selectOneMenu id="menu"
      value="immediateHandler.language"
      onchange="this.form.submit()" immediate="true"
      valueChangeListener=
        "#{immediateHandler.languageChanged}">
    <f:selectItems
        value="#{immediateHandler.languages}" />
  </h:selectOneMenu>
  <h:commandButton
      action="#{immediateHandler.action}"
      value="OK" />
```

Immediate


Das immediate-Attribut - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://localhost:8080/jsf-im-detail/pages/vc/
Kapitel 4: JavaServer ...
Das immediate-Attribut
Verwendung des immediate-Attributs
Anschrift
Sprache
Deutsch
OK
Fertig

Problemlösung durch `immediate`

- Konvertierung und Validierung der Komponente mit `immediate` schon in Phase 2
- Im Beispiel Value-Change-Listener aufgerufen:

```
public void languageChanged(ValueChangeEvent vce) {
  language = (String) vce.getNewValue();
  FacesContext.getCurrentInstance()
              .renderResponse();
}
```

- Sprache wird gesetzt
- Und es wird in Phase 6 gesprungen
- Für die *anderen* Komponenten gibt es *keine* Validierung/Konvertierung (`required` für Input)

Bean-Validierung mit JSR 303

Architekturproblem

- Validierung im GUI
- Und Validierung in der Persistenzschicht
- Muss man es zweimal machen ?

Bean Validation

- JSR 303
- Idee: Properties der Modell-Objekte werden mit Validierungsbedingungen annotiert
- Unabhängig von der Schicht kann validiert werden
- Enthalten in Java EE 6
- Und damit im Glassfish

Constraint-Annotationen des JSR 303

Annotation	Beschreibung
`@AssertFalse`	Property muss `false` sein.
`@AssertTrue`	Property muss `true` sein.
`@DecimalMax`	Property kleiner oder gleich Wert von `value`.
`@DecimalMin`	Property größer oder gleich Wert von `value`.
`@Digits`	Property darf nicht mehr Vor- und Nachkommastellen haben, als in `integer` und `fraction` angegeben..
`@Future`	Property muss Datum in der Zukunft sein.
`@Max`	Property kleiner oder gleich Wert von `value`.
`@Min`	Property größer oder gleich Wert von `value`.
`@NotNull`	Property darf nicht `null` sein.
`@Null`	Property muss `null` sein.
`@Past`	Property muss Datum in der Vergangenheit sein.
`@Pattern`	Property muss Pattern in `regexp` entsprechen.
`@Size`	Größe des Property muss zwischen `min` und `max` liegen.

Beispiel JSF

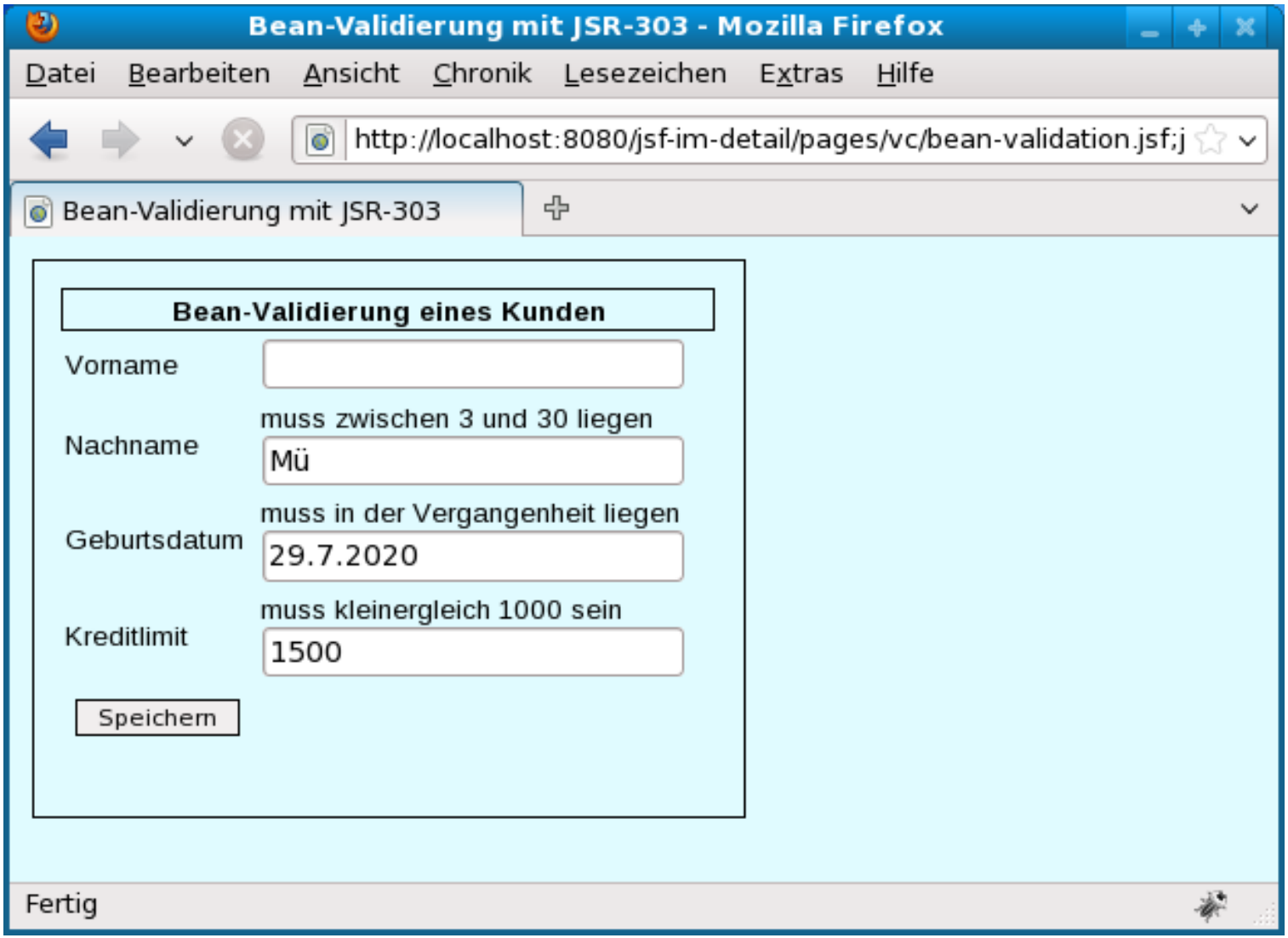
```
<h:panelGrid columns="2">
<h:outputLabel value="Vorname" />
<h:inputText id="vor"
    value="#{kundeHandler.kunde.vorname}" />
<h:outputLabel value="Nachname" />
<h:inputText id="nach"
    value="#{kundeHandler.kunde.nachname}" />
<h:outputLabel value="Geburtsdatum" />
<h:inputText id="gebu"
    value="#{kundeHandler.kunde.geburtstag}">
  <f:convertDateTime pattern="dd.MM.yyyy" />
</h:inputText>
<h:outputLabel value="Kreditlimit" />
<h:inputText id="limit"
    value="#{kundeHandler.kunde.kreditlimit}" />
<h:commandButton action="#{kundeHandler.speichern}"
    value="Speichern" />
```

- Nichts besonderes im GUI

Beispiel Java

```
public class Kunde {

  @NotNull
  private String vorname;

  @Size(min = 3, max = 30)
  private String nachname;

  @Past
  private Date geburtstag;

  @Min(0)
  @Max(1000)
  private Integer kreditlimit;
  ...
```

Bean-Validierung in Aktion



Eigene Constraints

- Bean-Validation besitzt API, um eigenen Validierungs-Constraints zu definieren
- Diese können wie die vordefinierten verwendet werden
- Machen wir leider nicht . . . , steht aber im Buch

Fehlermeldungen

Informationen für Benutzer

- nicht geglückte Konvertierungen, Validierungen und andere Informationen müssen angezeigt werden
- JSF definiert dafür ein Verfahren:
  - interne Darstellung: Resource-Bundle und `FacesMessage`
  - Benutzeranzeige: `<h:message>`, `<h:messages>`

Standardmeldungen

- lokalisiert, als Ressource-Dateien (Abschnitt 4.7)
- Ressource-Dateien sind Properties-Dateien (java.util.Properties)
- und damit Schlüssel/Wert-Paare mit Gleichheitszeichen als Trenner
- JSF-Spec gibt Schlüssel vor, Werte nicht
- jeder Schlüssel auch mit Suffix `_detail` (für detaillierte Meldung)

javax/faces/Messages_de.properties

```
javax.faces.component.UIInput.CONVERSION = \
Konvertierungsfehler

javax.faces.component.UIInput.CONVERSION_detail = \
"{0}": Fehler beim Model-Update.

javax.faces.component.UIInput.REQUIRED = \
Validierungsfehler

javax.faces.component.UIInput.REQUIRED_detail = \
"{0}": Eingabe erforderlich.

javax.faces.component.UISelectOne.INVALID = \
Validierungsfehler

javax.faces.component.UISelectOne.INVALID_detail = \
"{0}": Wert ist keine gltige Auswahl.

javax.faces.validator.NOT_IN_RANGE = \
Validierungsfehler
```

Anzeige von Meldungen

- Unterscheidung zwischen Meldung für eine Seite
- und Meldung für eine Komponente
- `<h:messages>` für Seite
- `<h:message for="...">` für Komponente
- `<h:messages>` erlaubt boolesches Attribut `globalOnly`
  - `true`: Nur die Meldungen, die keiner Komponente zugeordnet sind
  - `false`: Alle Meldungen der Seite.

Tipp

Achtung

Während der Entwicklung empfiehlt es sich, ein `<h:messages>` in jeder Seite zu haben. Damit ist sichergestellt, dass Sie auch alle Fehler angezeigt bekommen.

Alternativ/zusätzlich mit JSF 2

```
<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>
```

Werte der Enum `javax.faces.application.ProjectStage` erlaubt

Eigene Meldungen

- einfach zu realisieren
- siehe Buch ;-)

Event-Verarbeitung

- MVC-basierte GUIs de-facto Standard
- Grundlage Events (sonst zu starke Kopplung von M, V und C)

Problem bei Web-Anwendungen:

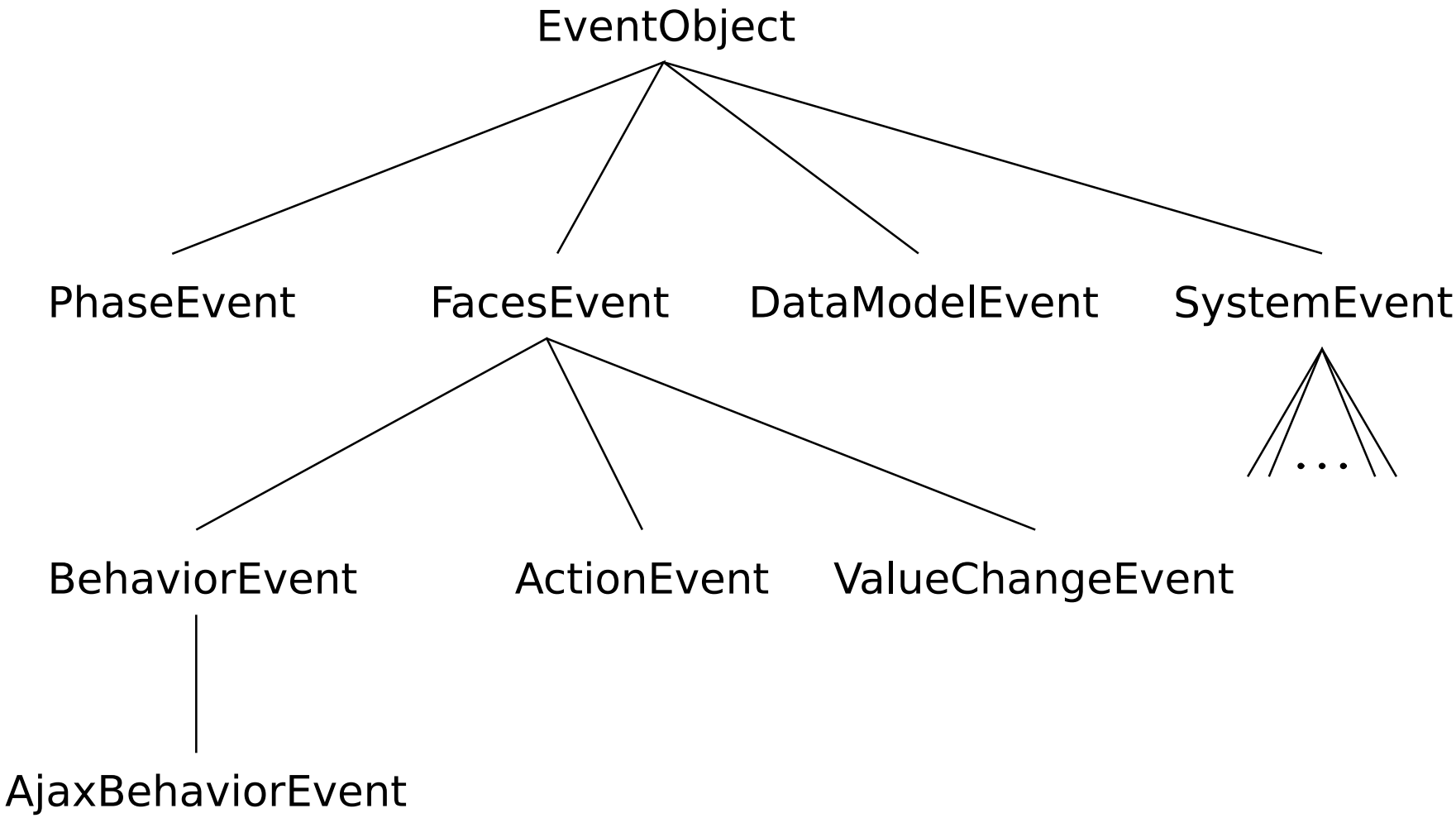
- Benutzer-Events im Client (Web-Browser)
- Verabeitung der Events (Controller) im Server
- JavaServer Faces realisieren trotzdem ein Event-basiertes MVC-Pattern
- und müssen dazu relativ viel Aufwand treiben, den man aber (oberflächlich) nicht sieht

JSF-Events und allgemeine Event-Verarbeitung

Allgemeines

- JSF-Events sind „normale“ Java-Events
- Vor JSF 2 vier Events: Value-Change-Events, Action-Events, Data-Model-Events, Phase-Events
- Mit JSF 2 System-Events und Ajax hinzugekommen
- Definition eigener Events möglich und bei eigenen Komponenten (Kapitel 7) evtl. auch sinnvoll
- `FacesEvent` abstrakte Oberklasse für die Events, die JSF-Komponenten werfen können: `ActionEvent` und `ValueChangeEvent`

Abbildung 4.19: Hierachie der JSF-Events



- `PhaseEvent` nicht an Komponenten, sondern an Bearbeitungsphasen (Abschnitt 4.1) gebunden
- `javax.faces.event` Package für alle Events außer `DataModelEvent`
- `DataModelEvent` nicht durch Benutzeraktionen sondern Änderungen der Datenkomponente
- System-Events z.B.: Pre-Render-View, Pre-Render-Component
- Java-Event-Muster mit *<name>Event* und *<name>Listener* gilt auch in JSF
- daher Listener-Interfaces `FacesListener`, `ActionListener`, `ValueChangeListener`, `PhaseListener` im Package `javax.faces.event`

Action-Events

Action-Events und Action-Listener

- Action-Events durch Benutzeraktionen (Befehlsmuster) ausgelöst: Drücken einer Schaltfläche, Klicken eines Links
- Action-Events werden durch Action-Listener bearbeitet (war klar)
- Zwei Arten von Action-Listener:
  - mit Rückgabewert, beeinflusst Navigation (Abschnitt 4.6)
  - ohne Rückgabewert, beeinflusst Navigation nicht (zwei Unterarten)
- Action-Listener in der Regel mit Programmlogik, ändern von Managed Beans und/oder Modellobjekten

Action-Listener-Methode

- Methode mit `ActionEvent`-Parameter und `void`-Rückgabe
- Name frei wählbar
- Attribut `actionListener` von `<h:commandButton>` und `<h:commandLink>`
- bereits im Tic-Tac-Toe, Kapitel 2 verwendet

Beispiel Action-Listener-Methode

```
...
<body>
<f:view>
  <h:form>
    <h:panelGrid columns="1">
      <h:outputText id="meldung" value="#{tttHandler.meldung}" />
      <h:panelGrid columns="3" styleClass="brett">
        <h:commandButton id="feld-0" image="#{tttHandler.image[0]}"
            actionListener="#{tttHandler.zug}" />
        <h:commandButton id="feld-1" image="#{tttHandler.image[1]}"
            actionListener="#{tttHandler.zug}" />
...
        <h:commandButton id="feld-8" image="#{tttHandler.image[8]}"
            actionListener="#{tttHandler.zug}" />
      </h:panelGrid>
      <h:commandButton value="Neues Spiel"
            action="#{tttHandler.neuesSpiel}"/>
    </h:panelGrid>
  </h:form>
</f:view>
</body>
</html>
```

Beispiel Action-Listener-Methode

```
public void zug(ActionEvent ae) {
  if (brett.isFertig())
    return;
  try {
    brett.setze(new Integer(ae.getComponent().getId().split("-")[1]))
    if (brett.isVerloren()) {
      meldung = "Herzlichen Glueckwunsch, Sie haben gewonnen";
      return;
    }
    brett.waehleZug();
    if (brett.isGewonnen()) {
      meldung = "Sie haben leider verloren";
    }
  } catch (Exception e) {
    log.info("Kein Spielerzug ausgefuehrt");
  }
}
```

Action-Methode

- noch einfacher: Name beliebig, keine Parameter, `String`-Rückgabe
- Attribut `action`
- wird in der Regel am häufigsten verwendet
- Bemerkung: Wert des `action`-Attributs ist JSF-EL-Ausdruck, kann also auch String-Konstante sein
- Rückgabewert wird für die Navigation (Abschnitt 4.6) verwendet
- Beispiele: `"success"`, `"failure"` für boolesche Ausgänge, Name von Prozessen (`"stammdateneingabe"`, `"persistieren"`, ...), ...

Action-Listener-Methode vs Action-Methode

Faustregel:

- nimm Action-Methode, falls navigiert werden soll
- nimm Action-Listener-Methode, falls möglichst einfach auf Action-Event und die das Action-Event erzeugende Komponente zugegriffen werden soll
- Problem: Was tun, wenn mehrere Methoden für ein Event zu registrieren sind? XML lässt nur maximal ein Attribut mit gleichem Namen zu.

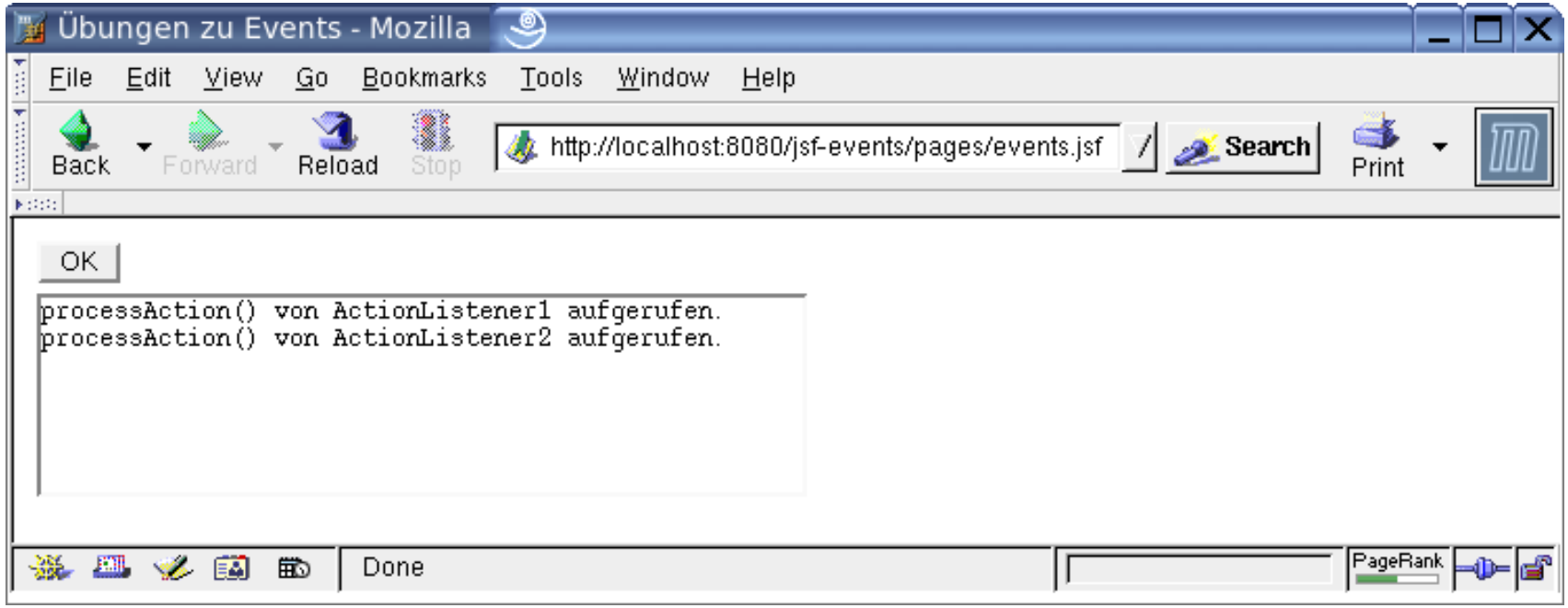
Action-Listener

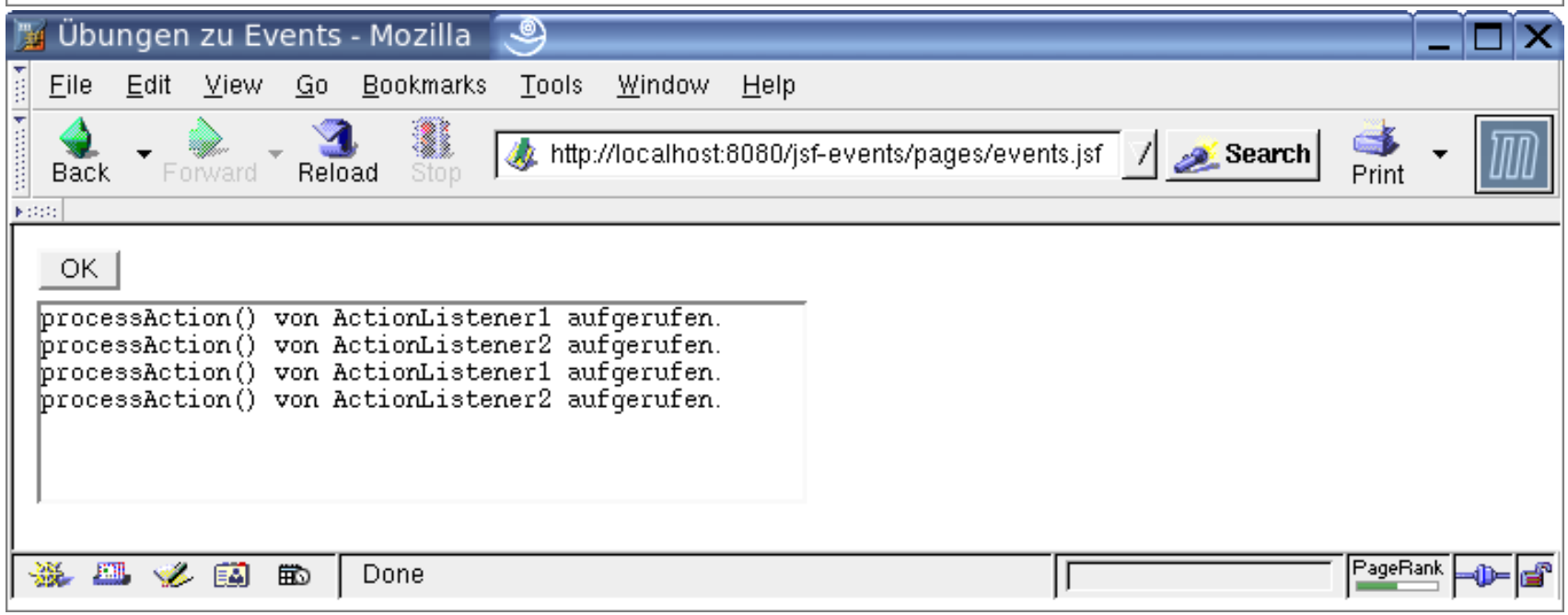
- Lösung: `<f:actionListener>`
- kann als geschachteltes Element mehrfach verwendet werden
- muss Interface `javax.faces.event.ActionListener` implementieren
- Beispiel mit zwei Action-Listenern in einem `<h:commandButton>`
- Achtung: Action-Listener greift (in Java) auf Komponente und in ihr enthaltenen Kompnenten zu!!!

Action-Listener II

```
<h:panelGrid id="panel" columns="1">
  <h:commandButton id="button" value="OK" action="success">
    <f:actionListener
        type="de.jsfpraxis.events.ActionListener1" />
    <f:actionListener
        type="de.jsfpraxis.events.ActionListener2" />
  </h:commandButton>
  <h:inputTextarea id="textarea" rows="6" cols="50"

      value="#{aeHandler.text}" />
</h:panelGrid>
```

```
public class ActionListener1 implements ActionListener {

  public void processAction(ActionEvent ae)
              throws AbortProcessingException {
    HtmlInputTextarea text = null;
    List components =
        ae.getComponent().getParent().getChildren();
    for (Iterator iter = components.iterator();
         iter.hasNext();) {
      UIComponent element = (UIComponent) iter.next();
      if (element.getId().equals("textarea")) {
        text = (HtmlInputTextarea) element;
      }
    }
    text.setValue(((String) text.getValue()).concat(
        "processAction() von ActionListener1 aufgerufen.
  }
}
```


Übungen zu Events - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/jsf-events/pages/events.jsf
Search
Print
OK
processAction() von ActionListener1 aufgerufen.
processAction() von ActionListener2 aufgerufen.
Done
PageRank


Übungen zu Events - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/jsf-events/pages/events.jsf
Search
Print
OK
processAction() von ActionListener1 aufgerufen.
processAction() von ActionListener2 aufgerufen.
processAction() von ActionListener1 aufgerufen.
processAction() von ActionListener2 aufgerufen.
Done
PageRank

Mit JSF 2 Parameter möglich

- Mit 2. Maintenance-Review der Unified EL Parameter möglich

```
public String action() { ... }
public String action(String text) { ... }

<h:commandButton value="action()"
      action="#{actionMethodParamHandler.action}" />
<h:commandButton value="action('Ein Text')"
      action="#{actionMethodParamHandler.action('Ein Tex
```

- Anwendungsfälle ?

Aufgabe

Schreiben Sie eine Action-Listener-Methode, die die Beschriftung einer Schaltfläche ändert. Wenn die Beschriftung „`einschalten`“ ist, soll sie nach Betätigung der Schaltfläche auf „`ausschalten`“ geändert werden und umgekehrt.

Aufgabe

In der Klasse `ActionListener.java` musste über die Komponenten iteriert werden, um die Komponente mit der richtigen Id zu finden. Eine Alternative ist die Verwendung der Methode `UIComponent.findComponent(String)`. Implementieren Sie diese Alternative.

Befehlskomponenten und das immediate-Attribut

Ändern des Bearbeitungszyklus

- Falls immediat gesetzt, Action-Listener nicht in Phase 5, sondern bereits in Phase 2
- Verwendung ?

```
<h:inputText id="data" value="#{iaHandler.data}"
             required="true" />
<h:commandButton action="#{iaHandler.speichern}"
             value="Speichern" />
<h:commandButton action="#{iaHandler.abbrechen}"
             value="Abbrechen" immediate="true"/>
```

- Sonst müssten Daten eingegeben werden, um abbrechen zu können

Value-Change-Events

ValueChangeEvent und ValueChangeListener

- in Anwendungen generell Interesse an Eingabewerteänderungen
  - Neueingabe, oder
  - Ändern eines Wertes
- z.B. bei Stammdatenänderung speichern nötig, ansonsten unnötig (Datenbankzugriff)
- „neu/geändert“ bedeutet bei Web-Anwendungen „von einem HTTP-Request zum nächsten“ (wenn man von AJAX absieht)
- Eingabekomponenten erzeugen `ValueChangeEvents` und `ValueChangeListener` können darauf registriert werden
- zwei Alternativen für Listener:
  - Methodenbindung über Attribut
  - Tag für eigene Listener-Klasse

Value-Change-Listener als Methode

- Methode muss einen Parameter vom Typ `ValueChangeEvent` haben
- und darf beliebig heißen
- `void` Rückgabe
- Beispiel: Erkennen einer Eingabeänderung und Anzeige dieser Änderung

```
<h:inputText value="#{vceHandler.text}"
    valueChangeListener="#{vceHandler.textChanged}" />
<h:commandButton value="OK" action="#{vceHandler.ok}" />
<h:outputText value="#{vceHandler.ausgabe}" />
```

```
public class ValueChangeEventHandler {

    private String text = "";
    private String ausgabe = "";
    private int reqNb = 0;
    private int reqNbChanged = 0;

    public void textChanged(ValueChangeEvent vce) {
        ausgabe = "Text hat sich geaendert. "
                + "War: '" + vce.getOldValue() + "'. "

                + "Ist: '" + vce.getNewValue() + "'.";
        reqNbChanged = reqNb;
    }

    public String ok() {
        if (reqNb != reqNbChanged)
            ausgabe = "";
        reqNb++;
        return "success";
    }
    ...
```

Beispiel Wertänderung

- Property `text` für Eingabe
- Property `ausgabe` für Ausgabe
- Änderung leicht zu erkennen und anzuzeigen, aber bei keiner Änderung muss die Anzeige der (letzen) Änderung wieder entfernt werden
- daher zwei Zähler, um Unterschied von Request und Request mit Änderung zu erkennen

Zweites Beispiel mit „teurer“ Berechnung

- Berechnung der Fakultät großer Zahlen (oder nicht, falls keine neue Zahl gewählt)
- auch lehrreich: Drop-Down mit „nicht konstanter“ Anzahl von Auswahlalternativen

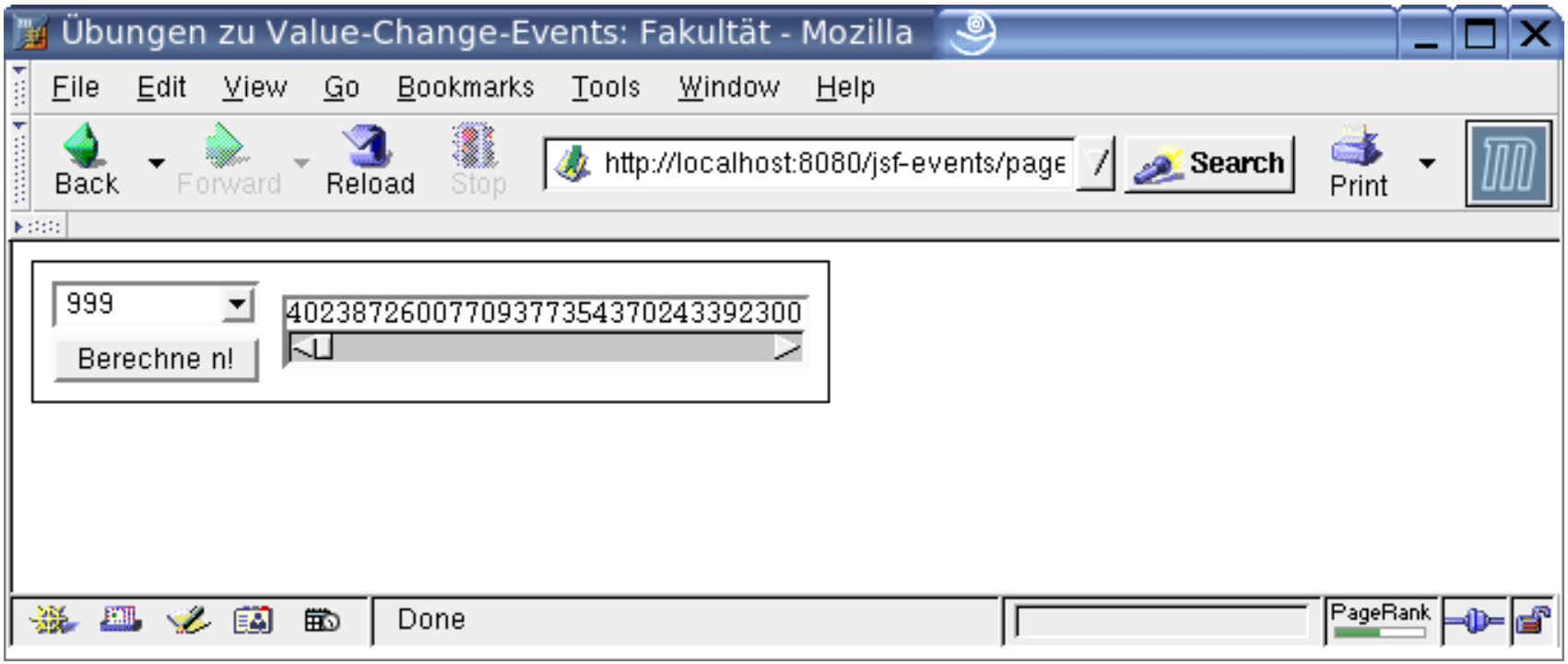
Beispiel Value-Change-Listener

```
<h:panelGrid columns="2" styleClass="borderTable">
  <h:panelGrid columns="1">
    <h:selectOneMenu style="width: 100%" value="#{fakHandler.n}"
       valueChangeListener="#{fakHandler.nChanged}">
      <f:selectItems value="#{fakHandler.arguments}" />
    </h:selectOneMenu>
    <h:commandButton value="Berechne n!" />
  </h:panelGrid>
  <h:panelGrid columns="1">
    <h:inputTextarea cols="30" rows="1"
       value="#{fakHandler.fakultaet}" />
  </h:panelGrid>
</h:panelGrid>
```

Value-Change-Listener-Methode

```
public class FakultaetHandler {

  private static final int N_MAX = 1000;
  private int n;
  private BigInteger fakultaet;

  public void nChanged(ValueChangeEvent vce) {
    fakultaet = Fakultaet.fakultaet(
        new BigInteger(vce.getNewValue().toString()));
  }

  /**
   * Liste von SelectItems potentieller Argumente
   */
  public List getArguments() {
    List l = new ArrayList();
    for (int i = 0; i < N_MAX; i++)
      l.add(new SelectItem(i, String.valueOf(i)));
    return l;
  }
}
```

Fakultätsberechnung



Große Unterschiede bei Betätigung der Schaltfläche bemerkbar, je nachdem, ob n geändert wurde oder nicht!

Value-Change-Event-Listener

- selbes Problem, wie bei Action-Event-Listenern: nur eine Listener-Methode, da nur ein Attribut `ValueChangeListener` erlaubt
- Lösung: Listener-Klasse, die Interface `javax.faces.event.ValueChangeListener` implementiert:

```
public interface ValueChangeListener extends FacesListener {
  public void processValueChange(
                javax.faces.event.ValueChangeEvent event)
      throws AbortProcessingException
}
```

- Beispiel

```
<h:inputText value="...
  <f:valueChangeListener type="de.jsfpraxis.MyListener1" />
  <f:valueChangeListener type="de.jsfpraxis.MyListener2" />
</h:inputText>
```

Aufgabe

Lösen Sie mit Hilfe eines Value-Change-Listeners die folgende Aufgabe: Wenn eine Benutzereingabe in ein `<h:inputText>` erfolgt ist, darf keine weitere ändernde Eingabe mehr erfolgen. Tipp: Die Java-seitige Komponente ist ein `HtmlInputText`. Diese kennt die Methode `setReadonly(boolean)`.

Aufgabe

Realisieren Sie die Aufgabe zur Texteingabeänderung ohne die Zähler in der Backing-Bean. Tipp: Verwenden Sie eine zusätzliche Managed-Bean mit Request-Scope.

Data-Model-Events

to do

Phase-Events

to do

System-Events

to do

Navigation

Navigation

- Frühe Web-Anwendungen mit in HTML hart verlinkten Seiten
- JSF für Unternehmensanwendungen, die in der Regel keine hart verlinkten Seiten nach diesem alten Muster beinhalten
- Ziel ist Navigation auf Basis *anwendungsbezogener Workflows*
- Dafür besitzen JSF eigenen *Navigation-Handler*:
  - Action-Events, bzw. deren Behandlung geben einen Wert zurück,
  - Der für die Navigation verwendet wird
  - Navigationsregeln in XML-Datei
  - Seit JSF 2.0 auch implizite Navigation, ohne Regeln
- Vorteile:
  - Navigation unabhängig von den JSF-Seiten, sondern abhängig vom *logischen* Ausgang eines Methodenaufrufs
  - Damit Seiten wiederverwendbar und besser wartbar
  - Gilt offensichtlich nicht für implizite Navigation

Syntax der Navigationsregeln

```
<navigation-rule>
    <from-view-id>
    <navigation-case>*
        <from-action>?
        <from-outcome>?
        <if>?
        <to-view-id>
        <redirect>?
```

Implizite Navigation

Direkte Angabe der View-Id

- Direkte Angabe der View-Id
  - In JSF-Seite
  - Return der Action-Methode
- In JSF-Seite
  - Attribut `action` bei `<h:commandButton>`, `<h:commandLink>`
  - Attribut `outcome` bei `<h:button>`, `<h:link>`

Beispiel

Direkt in JSF-Seite:

```
<h:commandButton value="..." action="ziel.xhtml"/>
```

Oder als Action-Methode:

```
<h:commandButton value="..."
                 action="#{handler.action}"/>
```

mit

```
public String action()  {
  ...
  return "ziel.xhtml";
}
```

Dateinamenserweiterung optional

- Man könnte also auch schreiben

```
<h:commandButton value="..." action="ziel"/>

public String action()  {
  ...
  return "ziel";
}
```

- Ist aber nicht ratsam. Verwechslungsgefahr mit Navigationsregeln

Verwenden?

- Für Anfänger sehr gut geeignet
- Für kleine Projekte ebenso
- Bei größeren Projekten Navigationsregeln in JSF-Konfigurationsdatei besser
- Achtung: JSF-Konfigurationsdatei geht vor impliziter Navigation

View-to-View-Regeln

Regel für „Seite zu Seite“

- View: alle Komponenten, die eine UI-Seite in JSF ausmachen
- jede View hat eine eindeutige Bezeichnung: die *View-Id* (siehe auch Abschnitt 4.9)
- View-Id ist kontextrelativer Pfad einer JSF-Seite
- View-Id kann als Quelle und Ziel einer Navigationsregel verwendet werden
- Navigationsregel durch `<navigation-rule>` definiert
- `<from-view-id>`: die Ursprungsseite dieser Regel

Regel für „Seite zu Seite“II

- mehrere `<navigation-case>` als Sprungverteiler:
  - `<to-view-id>`: obligatorisch
    Ziel dieser Navigationsregel
  - `<from-outcome>`: Ergebnis der Action-Methode (Konstante)
  - `<from-action>`: von welcher Action-Methode
    sinnvoll, falls mehrere Schaltflächen auf einer Seite

Beispiel

```
<navigation-rule>
  <description>
    Beispiel View-to-View-Regeln
  </description>
  <from-view-id>/pages/hauptseite.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>rot</from-outcome>
    <to-view-id>/pages/rot.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>gelb</from-outcome>
    <to-view-id>/pages/gelb.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>blau</from-outcome>
    <to-view-id>/pages/blau.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Beispiel (cont'd)

```
<h:outputLabel for="eingabe"
               value="Eingabe (rot/gelb/blau):" />
<h:inputText id="eingabe"
             value="#{naviHandler.eingabe}" />
<h:commandButton action="#{naviHandler.verteiler}"
                 value="Abschicken" />
```

Beispiel III

```
public String verteiler() {
  if (eingabe.equals("rot"))
    return "rot";
  else if (eingabe.equals("gelb"))
    return "gelb";
  else if (eingabe.equals("blau"))
    return "blau";
  else
    eingabe = "bitte nochmal";
  return "error";
}
```

keine Regel für "error": Default ist „selbe Seite nochmal anzeigen“

Regeln für mehrere Seiten

Wildcards

- Beispiel verzweigt zu den drei Seiten mit verschiedenen Hintergrundfarben
- Rückkehr zur Hauptseite benötigt drei Navigationsregeln mit verschiedenen `<from-view-id>`
- lässt sich mit Wildcards erheblich vereinfachen

Beispiel Wildcards

```
<navigation-rule>
  <description>
    Beispiel  Wildcard
  </description>
  <from-view-id>/pages/*</from-view-id>
  <navigation-case>
    <from-outcome>zurueck</from-outcome>
    <to-view-id>/pages/hauptseite.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

- Achtung: Überlappung mit alten `<from-view-id>`s
- ist zulässig und wird über die Reihenfolge der Regeln aufgelöst
- diese Regel muss also *nach* den drei anderen kommen

Regeln für Action-Methoden

Action-Methoden direkt als Trigger

```
<from-view-id>/pages/nav/hauptseite.xhtml</from-view-id>
<navigation-case>
  <from-action>#{naviHandler.rot}</from-action>
  <to-view-id>/pages/nav/ganzrot.xhtml</to-view-id>
</navigation-case>
```

- Action-Methode als EL-Ausdruck
- Wird genommen, falls Action-Methode aufgerufen

Bedingte Navigation

Bedingte Navigation (neu in JSF 2.0)

```
<from-view-id>...</from-view-id>
<navigation-case>
  <from-outcome>bedingt</from-outcome>
  <if>#{naviHandler.wert lt 100}</if>
  <to-view-id>/pages/nav/bedingt1.xhtml</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>bedingt</from-outcome>
  <if>#{naviHandler.wert ge 100}</if>
  <to-view-id>/pages/nav/bedingt2.xhtml</to-view-id>
</navigation-case>
```

- Property `wert` $<$ oder $\geq$ 100

Redirects

Redirect

- leeres Element `<redirect />`
- schickt Redirect-Antwort an den Broser, der initiiert neue Anfrage
- also: zwei Request/Response-Zyklen
- Vorteil: in der URL-Leiste steht korrektes URL der angezeigten View und „hinkt nicht um eins hinterher“
- Geht auch mit impliziter Navigation

Beispiel <redirect/>

```
<navigation-case>
  <from-outcome>redirect</from-outcome>
  <to-view-id>/pages/ganzrot.xhtml</to-view-id>
  <redirect />
</navigation-case>
```

Mit impliziter Navigation:

```
<h:commandButton
    action="ziel.xhtml?faces-redirect=true"/>
```

Oder als Action-Methode:

```
public String action()  {
  ...
  return "ziel.xhtml?faces-redirect=true";
}
```

Verweise auf Nicht-JSF-Seiten

Verweise auf Nicht-JSF-Seiten

immer möglich mit normalem <h:outputLink>

```
<h:outputLink value="http://www.jsfpraxis.de">
  <h:outputText value="Das Buch" />
</h:outputLink>
```

falls Schaltfläche gewünscht, zwei Alternativen

Alternative 1

JSF-Seite:

```
<h:commandButton action="das buch" value="Das Buch"/>
```

Navigation:

```
<navigation-case>
    <from-outcome>das buch</from-outcome>
    <to-view-id>/pages/das-buch.xhtml</to-view-id>
</navigation-case>
```

JSF-Zielseite:

```
<html>
  <head>
    <meta http-equiv="Refresh"
          content="0; URL=http://www.jsfpraxis.de" />
  </head>
```

Alternative 2

Action-Methode mit Benutzung des ExternalContext:

```
public String dasBuch() {
  FacesContext context = FacesContext.getCurrentInstance();
  ExternalContext ec = context.getExternalContext();
  try {
      ec.redirect("http://www.jsfpraxis.de");
  } catch (IOException e) {
      return "failure";
  }
  context.responseComplete();
  return "success";
}
```

View-Parameter und Lesezeichen

title

- ...

Die technische Sicht

Hintergründe

- nur UI-Komponenten, die das Interface `javax.faces.component.ActionSource` implementieren, können Action-Events auslösen
- in der Standardimplementierung sind dies `HTMLCommandButton` und `HTMLCommandLink`, beide Unterklassen von `UICommand`
- als JSF-Tag: `<h:commandButton>` und `<h:commandLink>`
- Verfahren der Event-Verarbeitung:
  - zuerst Action-Listener, die mit `<f:actionListener>` registriert wurden
  - dann Action-Listener, die mit Attribut `actionListener` registriert wurden
  - schließlich Action-Listener der Navigationskomponente, mit Attribut `action` registriert.
  - nur diese beeinflussen die Navigation, die beiden erstgenannten nicht

Aufgabe

Bei sehr vielen einzugebenden Daten werden diese häufig auf mehrere Seiten verteilt. Realisieren Sie eine solche Eingabe (mit beliebigen Daten) über drei Seiten, wobei

- *die erste Seite die Schaltflächen „Weiter“ und „Abbruch“,*
- *die zweite Seite die Schaltflächen „Weiter“, „Zurück“ und „Abbruch“*
- *und die dritte Seite die Schaltflächen „Zurück“, „Abbruch und „Abschluss“*

enthalten. Die Daten werden sinnvollerweise in einer Managed Bean gehalten.

Konfiguration

Die Servlet-Konfiguration

Das Faces-Servlet in web.xml

```
<!-- Servlet Definition -->
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
</servlet>

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

Kontextparameter in `web.xml`

```
<context-param>
  <param-name>Name</param-name>
  <param-value>Wert</param-value>
</context-param>
```

- Präfix: `javax.faces. ...`

Vordefinierte Variablen

Parameter	Beschreibung
`CONFIG_FILES`	Kontextrelative Liste von Dateien, falls mehrere JSF-Konfigurationsdateien verwendet werden sollen.
`DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE`	Flag zur Verwendung der Default-Zeitzone anstatt GMT in `DateTimeConverter`.
`DECORATORS`	Liste von Klassennamen (Type `TagDecorator`), durch Semikolon getrennt.
`DEFAULT_SUFFIX`	Liste von Alternativ-Suffixen für JSF-Dateien, durch Leerzeichen getrennt. Default ist „`.xhtml .jsp`“.

Vordefinierte Variablen (cont'd)

Parameter	Beschreibung
`DISABLE_FACELET_JSF_VIEWHANDLER`	Flag für den JSF 1.2 Kompatibilitäts- modus des View-Handlers.
`FACELETS_LIBRARIES`	Liste von Dateien für Facelets-Tag-Bibliotheken, durch Semikolon getrennt.
`FACELETS_BUFFER_SIZE`	Buffer-Größe des generierten `ResponseWriter`.
`FACELETS_REFRESH_PERIOD`	Zeitdauer in Sekunden, für die der Facelets-Compiler nach geänderten Facelets-Seiten sucht. Für `-1` wird nicht geprüft. Dies ist in der Regel für Produktionssysteme zu empfehlen.

Vordefinierte Variablen (cont'd)

Parameter	Beschreibung
`FACELETS_RESOURCE_RESOLVER`	Qualifizierter Klassenname des zu verwendenden `ResourceResolvers`.
`FACELETS_SKIP_COMMENTS`	Flag, das das Rendern eines XML-Kommentars in einer Facelets-Seite verhindert.
`FACELETS_SUFFIX`	Alternativer Suffix für Facelets-Seiten.
`FACELETS_VIEW_MAPPINGS`	Liste von Resource-Patterns, durch Semikolon getrennt. Passende Ressourcen werden als Facelets-Seiten interpretiert. Beispiel: „`/pages/*;*.xhtml`“.

Vordefinierte Variablen (cont'd)

Parameter	Beschreibung
`FULL_STATE_SAVING_VIEW_IDS`	Liste von View-Ids, durch Komma getrennt, deren Zustand komplett, d.h. wie in JSF 1.2 gespeichert werden.
`INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`	Leere Strings in Eingabefeldern werden in `null` umwandelt.
`LIFECYCLE_ID`	Qualifizierter Name der zu verwendenden `Lifecycle`- Implementierung.
`PARTIAL_STATE_SAVING`	Flag zur Steuerung der partiellen Zustandsspeicherung.

Vordefinierte Variablen (cont'd)

Parameter	Beschreibung
`PROJECT_STAGE`	Aktueller Entwicklungsstand des Systems. Mögliche Werte: `Development`, `UnitTest`, `SystemTest`, `Production`.
`RESOURCE_EXCLUDES`	Liste von Dateinamenerweiterungen von Ressourcen, die nicht ausgeliefert werden dürfen. Default-Wert: `.class .jsp .jspx .properties .xhtml`.
`STATE_SAVING_METHOD`	Speicherort für den Komponentenbaum. Mögliche Werte sind `server` (Default) und `client`.

Vordefinierte Variablen (cont'd)

Parameter	Beschreibung
`SEPARATOR_CHAR`	Trennzeichen für Client-Ids, die mehrere Bestandteile haben. Default ist der Doppelpunkt, der laut Spezifikation nicht geändert werden darf.
`VALIDATE_EMPTY_FIELDS`	Validierung leerer Eingaben, falls `true`. Für den Wert `auto` wird bei Vorhandensein einer JSR-303-Implementierung `true`, sonst `false` gesetzt.
`DISABLE_DEFAULT_BEAN_VALIDATOR`	Flag, das die Verwendung einer JSR303-Implementierung steuert.

title

- ...

Die JSF-Konfiguration

title

- ...

XML-Konfiguration versus Annotationen

title

- ...