14. – 17. 09. 2009
in Nürnberg

Wissenstransfer
par excellence

Ganz vorne.

JSF Input Controls validieren gemäß WSDL

(R32 – 17.09.2009, 11:20-12:30 h)

Alexander Schwartz

Ikano Bank GmbH

Inhalt der Präsentation

- Wer ist Ikano Bank
- Wie sieht unsere Java-Architektur aus
- Welche Komponenten/Werkzeuge nutzen wir
- Eingabevalidierung am Beispiel Struts 1.x und JSF 1.2
- Eingabevalidierung mit JBoss Seam
- Eingabevalidierung mit XSDs/WSDLs
- Mechanismen von Seam zur Validierung
- Generische Pflichtfeldvalidierung
- Generische Validierung anhand von WSDLs
- Fazit

Unsere Herkunft

- Seit 1996 in Deutschland
- Über 1 Mio. Kunden
- Zusammenarbeit mit über 3000 Handelspartnern

IKANO®

1943: von Ingvar Kamprad als Teil von IKEA gegründet.

Seit 1988: unabhängiger Konzern, im Eigentum der Familie Kamprad

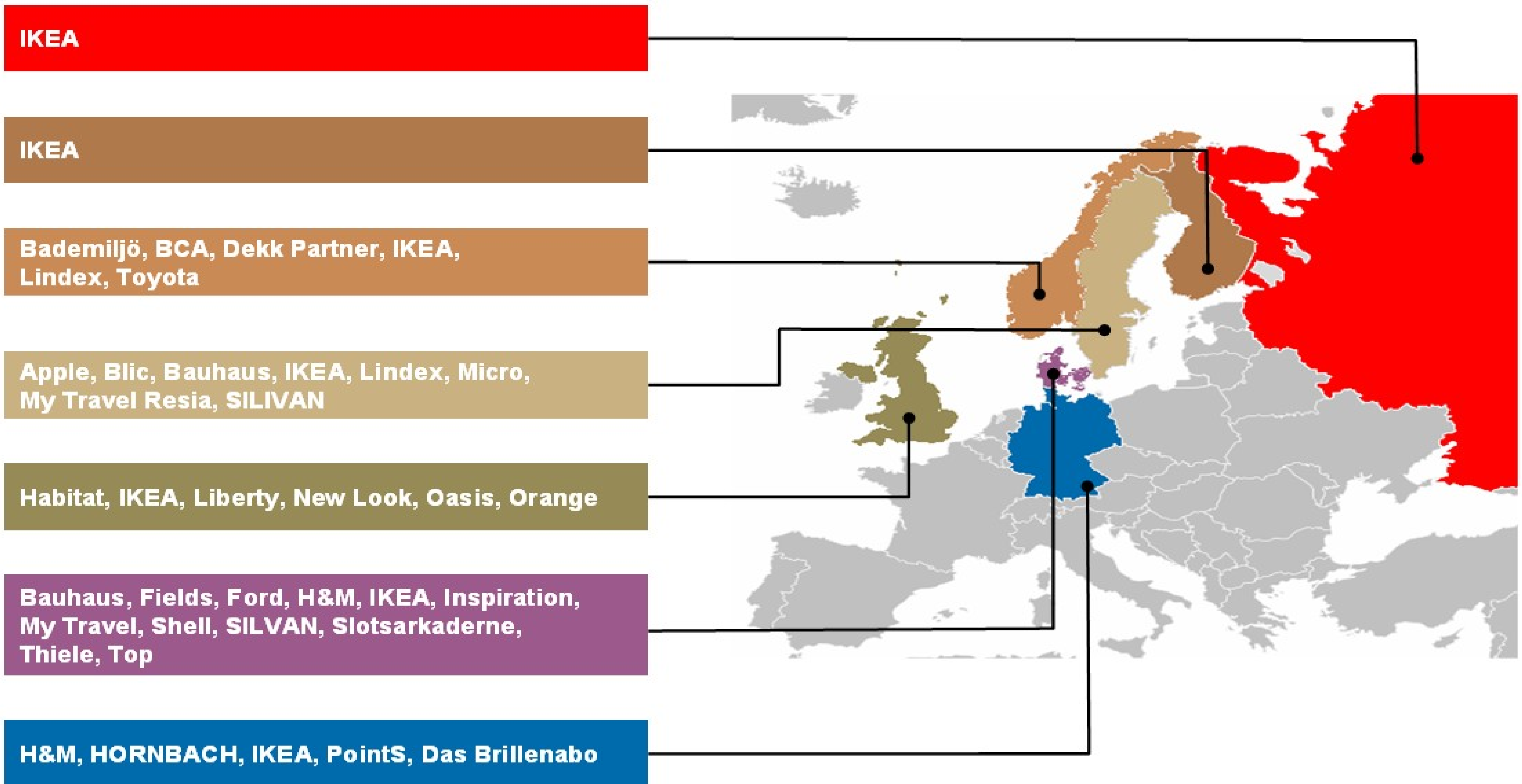
Ikano besitzt Unternehmen in den Bereichen Finanzdienst-leistungen, Immobilien und Einzelhandel.

1943: von Ingvar Kamprad als Versandhandel gegründet

2009: 285 Einrichtungshäuser in 36 Ländern Umsatz 21 Mrd. Euro

IKANO BANK

Internationale Präsenz



Wer ist Ikano Bank Deutschland?

- Spezialist für Absatzfinanzierung
- Produkte: Finanzierungen, Kundenkreditkarten, Kreditkarten
- Partner: IKEA, H&M, Hornbach, Das Brillenabo, Automeister, point S
- Über 3.000 Geschäfte, 1,1 Millionen Kundenkarten
- Beantragung: im Markt, per Post, per Internet

Wer bin ich?

- Teamleiter JAVA Entwicklung
- Architekt für die Java-Anwendungen rund um Beantragung von Krediten und Karten und Internet Banking
- Einstieg in JAVA in 2003
- Zunächst: Struts 1.x und EJB
- Online Banking der Direkt-Bank 1822direkt
- Später bei der Ikano Bank: Oracle und PL/SQL Vertiefung
- Aktuell: JSF 1.2 RI + RichFaces + SEAM + Spring Webservices

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Was für Anwendungen haben wir?

- Kartenmanagement-System auf Basis Oracle PL/SQL und Forms
  - Wird im Callcenter genutzt
  - Verwaltet Karten- und Finanzierungskonten
  - Stellt die Ratenzahlung der Kunden sicher
- Dokumenten-Management-System
  - Workgroup-System für eingehende Korrespondenz
  - Archiviert die Verträge mit den Kunden
  - Archiviert die tägliche Kommunikation mit den Kunden
- Antragssystem und Kundenverwaltung im Markt/im Internet
  - Deckt den vorvertraglichen Prozess ab bis zur Genehmigung und Unterschrift des Kunden
  - Wird in allen Märkten als Web-Anwendung genutzt
  - Webservice-Schnittstelle für Partner geplant

IKANO BANK

Antragssystem: Idee

- Legitimation von Mitarbeitern beim Partner
- Bei Bedarf Rücksetzen des Passworts
- Antragseingabe durch
  - Kunden am SB-Terminal im Markt,
  - Patner-Mitarbeiter im Markt
  - Ikano-Bank-Mitarbeiter in Wiesbaden
  - Kunden im Internet
- Abdecken des Angebots- und Genehmigungsprozesses
- Zugriff auf Vertragsinformationen
- Pflege von Kunden-Stammdaten, Aktionsplanung, Reporting

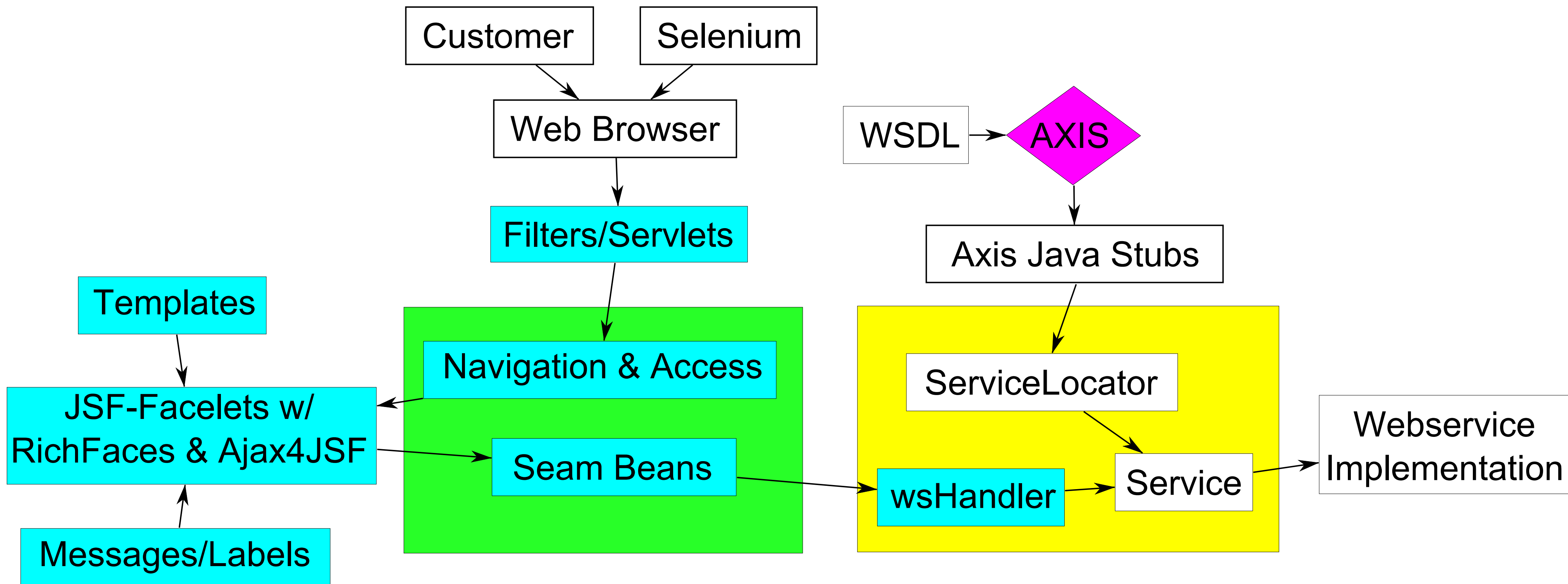
Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Antragssystem: Technische Details

- JSF-basierte Web-Anwendung mit AJAX
- Webservice Backend
- Mehrsprachig
- Unterstützung von Themes für verschiedene CIs für verschiedene Partner
- Antragsformular als Metadaten in der Datenbank konfiguriert
- Rollen- und Berechtungskonzept um interne/externe Sicht und verschiedene Partnersichten abdecken zu können.

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Antragssystem: Architektur



Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

IKANO BANK

Warum welche Komponenten?

- JSF
  - (Java) Standard für komponentenbasierte Webanwendungen
  - Angemessen für die fachlichen Anforderungen
- Facelets
  - Templating-Mechnanismus
  - Dekorieren von Elementen möglich
- RichFaces
  - Steuert AJAX/Web 2.0 View-Komponenten bei

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Warum welche Komponenten?

- Seam
  - Erweitert insbesondere Controller und Modell, z.B. Konversationen, eine erweiterte Expression Language, Rollenbasiertes Zugriffmodell, Konfiguration durch Annotationen
  - Unterstützung von PageFlows für komplexere Navigation und die Behandlung des Zurück-Buttons
- Ikano Bank Erweiterungen
  - In Standard-Usecases werden die Validierungen der Formulare aus der WSDL gezogen
  - Antragsformulare werden als Metadaten aus der Datenbank gelesen inkl. Validierungen, Positionierungen und Semantik.

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Welche Werkzeuge werden genutzt?

- Eclipse
  - Java IDE, erweiterbar durch Plugins
  - Open Source
  - JSF-Editor ist enthalten
- JBoss IDE
  - Plug-in für Eclipse zur Unterstützung von Seam und RichFaces
  - Plug-in für JPDL Pageflows
- Selenium
  - Automatisierte Tests für Web-Frontends
  - Kann auch von einer Fachabteilung bedient werden
- + Maven, JUnit

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Eingabevalidierung im Frontend

- Architekturentscheidung wurde Anfang 2008 bei Ikano Bank getroffen

- Was es schon gab
  - Eingabevalidierung in Struts 1.x
  - Eingabevalidierung in JSF 1.2
  - Eingabevalidierung in Seam mit Hibernate Validator

- Was seit dem dazugekommen ist
  - Metawidget (erlaubt ähnliche Annotationen wie Hibernate Validator)

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Validierung bei Struts 1.x

- Struts ist ein MVC Framework
- Validierungen werden separat von Modell und View hinterlegt in einer Validator-Konfiguration
- Globale Konstanten können definiert werden
- Jedes Formular bekommt eine separate Konfiguration
- Validierung gleichzeitig client- und serverseitig
- Fehlermeldungen können sich auf den Feldnamen beziehen und erlauben Parameter
- Abhängige Validierung möglich
- Erweiterung durch eigene Validatoren möglich (habe ich aber nie gemacht)

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Validierung bei Struts 1.x: Beispiel 1

```
<global>
  <constant>
    <constant-name>emailMaxLength</constant-name>
    <constant-value>80</constant-value>
  </constant>
  ...
</global>
```


IKANO BANK

Validierung bei Struts 1.x: Beispiel 2

```
<form name="/doRegistration">
  <field
    property="eaddAddress"
    depends="required,maxlength,email">
    <arg0 key="label.email" />
    <arg2
      key="${var:maxlength}"
      name="maxlength"
      resource="false" />
    <var>
      <var-name>maxlength</var-name>
      <var-value>${emailMaxLength}</var-value>
    </var>
  </field>
</form>
```

Validierung bei JSF 1.2

- JSF als komponentenbasiertes Framework
- Validierungen werden direkt im View eingetragen
- Jede Page eine separate Validierungs-Konfiguration
- Validierung nur serverseitig
- Fehlermeldungen können sich auf den Feldnamen beziehen und erlauben Parameter
- Abhängige Validierung nicht möglich, nur mehrfache Validierungen pro Eingabefeld

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

IKANO BANK

Validierung bei JSF 1.2: Beispiel

```
<h:inputText id="eaddAddress" value="#{bean.eaddAddress}" required="true">
      <f:validateLength maximum="#{valConstants.emailMaxLength}"/>
</h:inputText>
```

Validierung bei JSF 1.2: Was fehlt

- Validierung wird im View hinterlegt und muss konsistent gehalten werden
- Validatoren werden nur ausgelöst, wenn auch ein Wert eingegeben wurde

IKANO BANK

Ausgangssituation bei JSF

- Vermischung von Anzeige und Validierung
  - offen
- Pflichtfeld-Eigenschaft ist kein Validator
  - offen
- Validierungsregeln pro View definiert
  - offen
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - offen

IKANO BANK

Validierung bei JSF mit SEAM

- SEAM erweitert JSF um vieles, das fehlt, unter anderem eine weitergehende Validierung
- Validatoren können als Annotationen im Model hinterlegt werden
- Mehrere Formulare, die die gleichen Backing Beans benutzen, nutzen die gleichen Validierungsregeln
- SEAM greift dabei auf Hibernate Annotationen für Validierungen zurück
- Wird im Projekt Hibernate für eine direkte Persistenz eingesetzt, können die dort getroffenen Annotationen (enge Kopplung vorausgesetzt) weiterverwendet werden

Validierung bei JSF mit SEAM: Beispiel

```
<s:validateAll>
   <h:inputText id="eaddAddress" value="#{bean.eaddAddress}" required="true" />
   <h:inputText ... />
   ...
</s:validateAll>

public class Registration {
    private String eMail;
    @Length(max=80)
    public String getEMail() { return eMail; }
    public void setEMail(String eMail) { this.eMail = eMail; }
}
```


IKANO BANK

Validierung bei JSF mit SEAM: Was fehlt/gefällt

Was fehlt:

- Required-Attribute müssen weiterhin im View eingetragen werden
- Abhängige Validierungen sind zunächst nicht dabei

Was gefällt:

- Zusätzliche Validatoren lassen sich einfach entwickeln
- Validatoren können nur einfache Parameter haben, in Strings kann man allerdings auch EL-Ausdrücke angeben

Die andere Seite: Validierung in XSD/WSDL

- Webservices liefern in unserer Architektur die Backendfunktionen
- Validierungen auf Attribut-Ebene
- Ausdrucksstark
- Zusammen mit natürlichsprachlicher Dokumentation innerhalb der WSDL beschreiben sie den Vertrag zwischen Frontend und Backend
- Im Ansatz „Contract First“ wird dies zunächst vom Frontend- und Backend-Team zusammen spezifiziert und später weiterentwickelt

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

IKANO BANK

Validierung mit XSD: Zeichenketten

```
<xs:element name="eaddAddress">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-z])*"/>
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
      <xs:length value="8"/>
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```


IKANO BANK

Validierung mit XSD: Zahlen

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```


IKANO BANK

Validierung XSD/WSDL: Was fehlt/gefällt

Was gefällt:

- Alle Standardvalidierungen sind vorhanden
- Die Validierungen können vom Backendteam gepflegt und verstanden werden
- Sie sind maschinell auswertbar, und können bei jedem Webservice-Aufruf validiert werden, sowohl für Anfragen als auch für Antworten
- Wahlweise in Produktion für Performanceoptimierung deaktiviert

Was fehlt:

- Weitergabe der Validierungen zum Anwender ins Frontend

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Zwischenstand

Was haben wir gesehen?

- Validierung im Frontend bei Struts 1.x, JSF, JSF+SEAM
- Validierung an der Schnittstelle zum Backend mit WSDL

Was ist das Ziel?

- Validierung aus den Views entfernen
- Zufüttern aus der Schnittstelle des Backends (WSDL), ggf. Ergänzungen aus dem Modell

Wie geht es weiter?

- Vorstellung der Implementierung bei SEAM im Detail
- Erweiterung der SEAM-Funktionen zur Übernahme von WSDL Validierungen

Wie SEAM die Hibernate Validierungen findet

- <s:validateAll /> Tag markiert die Eingabeelemente
- Abgeleitet von JSF Renderer
- Bei jedem Aufbau des Views (doEncodeChildren)
- JSF erlaubt ein durchsuchen der Komponenten nach Eingabefeldern (EditableValueHolder)
- SEAM erstellt einen dekorierten EL Context/Resolver
- Über ein durch den Decorator abgefangene „Set“ findet SEAM das Ziel der Value-Expression, die dem Eingabeelement zu Grunde liegt
- Statt des „Set“ wird dann eine Validierung des Eingabewerts durchgeführt, die gefundenen Fehler werden dann in FacesMessages umgewandelt

SEAM/Hibernate Validierung 1

- Durchsuchen der Child Elemente

```
private void addValidators(List children)
{
    for (Object child: children)
    {
        if (child instanceof EditableValueHolder)
        {
            EditableValueHolder evh =  (EditableValueHolder) child;
            if ( evh.getValidators().length==0 )
            {
                evh.addValidator( new ModelValidator() );
            }
        }
        addValidators( ( (UIComponent) child ).getChildren() );
    }
}
```

SEAM/Hibernate Validierung 2

- Dekorierter Context/Resolver

```
public InvalidValue[] validate(ValueExpression valueExpression, ELContext
elContext, Object value)
{
   ValidatingResolver validatingResolver = new
                            ValidatingResolver(elContext.getELResolver());
   ELContext decoratedContext = EL.createELContext(elContext,
                                                    validatingResolver);
   valueExpression.setValue(decoratedContext, value);
   return validatingResolver.getInvalidValues();
}
```


IKANO BANK

SEAM/Hibernate Validierung 3

- Dekorierter Context/Resolver

```
public void setValue(ELContext context, Object base, Object property, Object
 value)
    {
        if (base != null && property != null)
        {
            context.setPropertyResolved(true);
            invalidValues = getValidator(base).
                        getPotentialInvalidValues(property.toString(), value);
        }
    }
```


IKANO BANK

Zwischenfazit SEAM Validierung

- SEAM findet alle Validierungen, die direkt an den EL-Attributen der Eingabeelemente hängen

Notwenige Erweiterungen

- Required Attribut soll automatisch vergeben werden
- Validatoren zusätzlich aus den aus der WSDL generierten Klassen ableiten, falls diese per Delegator-Pattern verwendet werden

Ausgangssituation bei JSF + SEAM

- ~~Vermischung von Anzeige und Validierung~~ –
  - Teilweise gelöst (außer Pflichtfeld-Validierung)
- Pflichtfeld-Eigenschaft ist kein Validator
  - offen
- ~~Validierungsregeln pro View definiert~~
  - Teilweise gelöst (außer Pflichtfeld-Validierung)
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - offen

IKANO BANK

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Erweiterung 1: Setzten von Required

Lösungsansatz:

- Wenn ein Validator eines Objekts für den Eingabewert null einen Fehler liefert, so wird das Required-Attribut gesetzt.
- Für eine Checkbox ist es stattdessen ein Boolean

Wo einbauen?

IKANO BANK

Erweiterung 1: Setzten von Required

```
for (javax.faces.validator.Validator v : uc.getValidators()) {
   if (!uc.isRequired()) {
     try {
       if (uc instanceof UISelectBoolean) {
         v.validate(FacesContext.getCurrentInstance(), uc, Boolean.FALSE);
       } else {
         v.validate(FacesContext.getCurrentInstance(), uc, null);
       }
     } catch (ValidatorException e) {
       /* sometimes we can't find the property i.e. when it is in a loop
        * or uses a temporary variable. Then we will see a
        * PropertyNotFoundException and will ignore it.  */
       if (!(e.getCause() instanceof PropertyNotFoundException)) {
         uc.setRequired(true);
       }
     }
   }
```


IKANO BANK

Fazit: Ausgangssituation bei JSF + SEAM + Erweiterung Pflichtfeld

- ~~Vermischung von Anzeige und Validierung~~
  - Vollständig gelöst
- ~~Pflichtfeld-Eigenschaft ist kein Validator~~
  - Vollständig gelöst
- ~~Validierungsregeln pro View definiert~~
  - Vollständig gelöst
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - offen

Erweiterung 2: Finden von WSDL Validierungen

- WSDL Objekte werden (bei uns) nicht direkt per EL angesteuert, sondern indirekt per Delegator
- Dadurch können wir die Beans besser passend zur UI anpassen, ggf. auch Attribute nicht ans Frontend weitergeben
- Dadurch reicht aber auch ein dekorierter EL-Resolver/Context nicht aus
- Alternative: die WSDL Objekte werden (zur Laufzeit) instrumentiert, um die Verbindung zwischen Value Bindings und den darunterliegenden WSDL Objekten aufzuzeichnen

IKANO BANK

Lösungsansatz: Aufzeichnung von Zugriffen

- Instrumentierung des Delegators
- Aktivieren der Aufzeichnung
- Eine Get-Operation wird auf das Value-Bindung ausgeführt (Annahme: eine Get-Operation verändert den Zustand des Modells nicht)
- Die Zugriffe werden inklusive der Get-Methoden auf den aus der WSDL generierten Klassen ausgeführt
- Ein Inteceptor fängt das Get ab, schlägt die Validierungen in der WSDL nach, liefert einen Dummy-Wert zurück
- Die aufgezeichneten Validierungen werden dann als Validierungen zum Eingabeelement hinzugefügt

IKANO BANK

Schritt 1: Instrumentierung des Delegators

- Mit Bytecode-Instrumentierung können Klassen zur Laufzeit erzeugt verändert werden
- Standardbibliotheken sind z.B. javassist und CGLIB, Java bietet auch Funktionalitäten
- In diesem Fall wurde CGLIB verwendet, da dort im Vergleich zu den Standard-Java-Funktionen als Basis die aus der WSDL generierten Klassen dienen sollen
- Es wird ein MethodInterceptor angehängt, mit dem alle Zugriffe auf Methoden bei Bedarf protokolliert werden sollen
- Dies ist als Factory implementiert, um die Bedienung zu vereinfachen

IKANO BANK

Schritt 1: Instrumentierung

```
public static Object getInstance(Class<?> clazz, boolean
 requiredCheckEnabled) {
  WsdlValidatorInterceptor interceptor = new WsdlValidatorInterceptor(
      requiredCheckEnabled);
  Enhancer e = new Enhancer();
  e.setSuperclass(clazz);
  e.setCallback(interceptor);
  return e.create();
}
```


IKANO BANK

Schritt 1: Warum manchmal ohne RequiredCheck

- In der Regel sollen alle Validierungen übernommen werden
- Wenn aber z.B. Eingabefelder für mehrere Telefonnummern auf einer Seite angezeigt werden sollen, bei denen die komplette Telefonnummer optional ist, innerhalb des Telefonnummern Typs aber Vorwahl und Rufnummer Pflicht sind, dann passt die automatische Übernahme von Validierungen auf Feldebene nicht
- In diesem Fall muss die Validierung ausprogrammiert werden

Schritt 2: Aktivierung der Aufzeichnung

- Die Aufzeichnung muss bei allen Instanzen greifen, die beim späteren Get angesprochen werden sollen
- Die Instanzen selbst sind noch unbekannt
- Daher: setzen einer ThreadLocal Variable

IKANO BANK

Schritt 2: Aktivierung der Aufzeichnung

```
public class WsdlValidatorInterceptor implements MethodInterceptor,
    Serializable {

  private static ThreadLocal<List<ValidationDetails>> vals = new
   ThreadLocal<List<ValidationDetails>>();

  public static void record() {
    vals.set(new ArrayList<ValidationDetails>());
  }

  ...

}
```


IKANO BANK

Schritt 3: Ausführen der Get-Method

- Die Aufzeichnung muss bei allen Instanzen greifen, die beim späteren Get angesprochen werden sollen
- Die Instanzen selbst sind noch unbekannt
- Daher: setzen einer ThreadLocal Variable

IKANO BANK

Schritt 3: Ausführen der Get-Method

```
ValueExpression ve = component.getValueExpression("value");
if (ve != null) {
  WsdlValidatorInterceptor.record();
  Object o = ve.getValue(facesContext.getELContext());
  ...
}
```


IKANO BANK

Schritt 4: Abfangen der Get-Methode und Nachschlagen der Validierungen

- Ist dies eine Get-Methode?
- Ist die Aufzeichnung aktiviert?
- Ableiten der WSDL auf Basis des Pakets
- Suchen in der WSDL via XPATH anhand von Klassen- und Methodenname

IKANO BANK

Schritt 4: Abfangen der Get-Methode ...

```
public Object intercept(Object obj, Method method, Object[] args,
    MethodProxy proxy) throws Throwable {
  Object o = proxy.invokeSuper(obj, args);
  // if this is a getter
  if (method.getName().startsWith(GET) && args.length == 0
      && Modifier.isPublic((method.getModifiers()))) {

    ....

    }
}
```


IKANO BANK

Schritt 4: ... Suchvorbereitung ...

```
String file = obj.getClass().getName();
file = file.replaceAll("\\.[^\\.]*$", "");
file = file.replaceAll("^.*\\.", "");
doc = builder.parse(this.getClass().getResourceAsStream(
    "/" + file + ".wsdl"));

// find element matching the request object/method
String field = method.getName().substring(GET.length(), GET.length() + 1)
    .toLowerCase(Locale.ENGLISH)
    + method.getName().substring(GET.length() + 1);

String clazz = method.getDeclaringClass().getCanonicalName();
clazz = clazz.replaceAll("\\$.*$", "");
String request = clazz.replaceAll(".*\\.", "");
```


IKANO BANK

Schritt 4: ... Suche via XPATH

```
XPathExpression expr = xpath.compile("//xs:element[@name='" + request
    + "']//xs:element[@name='" + field + "']");
NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);

if (nodes.getLength() == 0) {
  expr = xpath.compile("//xs:complexType[@name='" + request
      + "']//xs:element[@name='" + field + "']");
  nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
}
```

Schritt 5: Lesen der Validierungen...

```
     XPathExpression expr = xpath
                 .compile("./xs:simpleType/xs:restriction");
            NodeList restrictions = (NodeList) expr.evaluate(node,
                XpathConstants.NODESET);

...
            for (int j = 0; j < restrictions.getLength(); j++) {
              Node restriction = restrictions.item(j);
              String base = restriction.getAttributes().getNamedItem("base")
                  .getTextContent();

              if ("xs:string".equals(base)) {
                expr = xpath.compile("./xs:pattern/@value");
                String regex = (String) expr.evaluate(restriction,
                    XpathConstants.STRING);
                validations.add(createRegexValidation(o, regex, method));
...
```


IKANO BANK

Schritt 5: ... Bereitstellen als Hib. Validator

```
private ValidationDetails createRegexValidation(Object o, String regex,
      Method m) {
    ValidationDetails mv = new ValidationDetails();
    PatternValidator lv = new PatternValidator();
    PatternImpl l = new PatternImpl();
    l.setRegex(regex);
    StringBuilder sb = new StringBuilder();
    sb.append(m.getDeclaringClass().getCanonicalName()).append(".");
    sb.append(cutMethodNamePrefix(m.getName())).append(".regex");

    if (!Messages.instance().get(sb.toString()).equals(sb.toString())) {
      l.setMessage("{" + sb.toString() + "}");
    }
    lv.initialize(l);
    mv.setValidator(lv);
    mv.setAnnotation(l);
    mv.setValue(o);
    return mv;
  }
```


IKANO BANK

Schritt 6: Validieren des Eingabewerts

```
 List<Validator<?>> validators = WsdlValidatorInterceptor.retrieve(o,
    mia);
for (Validator<?> v : validators) {
  if (!v.isValid(value)) {
    String message = mia.getAnnotationMessage(v);
    message = mia.interpolate(message, v, null);
    throw new ValidatorException(new FacesMessage(
        FacesMessage.SEVERITY_ERROR, message, null));
  }
}
```


IKANO BANK

Was es abrundet

- Implementierung der verschiedenen Hibernate Validatoren passend zu den XSD restrictions
  - Fleißarbeit
- Caching für das Parsen der WSDL
  - Key: Klasse und Element
  - Value: Eine Collection mit Validierungen
- Individuelle Fehlermeldungen für Regex-Validierungen
  - Default: „Das Eingabemuster muss „{regex}“ entsprechen“
  - Alternative: Via Resource Bundle (messages) spezifische Meldungen hinterlegen
  - Key: Package + Klasse + Element + Validator
  - Gefahr, dass Validierung und Meldung nicht mehr zueinander passen

Was man noch machen könnte

- XSD Validierungen reichen nicht für komplexe Validierungen, z.B. „ist eine E-Mail-Adresse“
  - Ausweichen auf xs:annotation mit xs:appinfo
  - Dort können spezifische Informationen (auch im XML-Format) mit ggf. eigenem Namespace hinterlegt werden
  - Wir nutzen es, um Aufrufe zu markieren, deren Ergebnisse gecached werden können
- Serialisierung
  - Instrumentierung mit CGLIB verhindert zunächst Deserialisierung in einer anderen JVM wg. dynamischen Klassen, damit ist keine Session-Replikation möglich
  - Durch Implementieren von writeReplace/readResolve ist es aber möglich (bei uns derzeit aber nicht notwendig)

IKANO BANK

Fazit: Ausgangssituation bei JSF

- Vermischung von Anzeige und Validierung
  - offen
- Pflichtfeld-Eigenschaft ist kein Validator
  - offen
- Validierungsregeln pro View definiert
  - offen
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - offen

IKANO BANK

Fazit: Ausgangssituation bei JSF + Seam

- ~~Vermischung von Anzeige und Validierung~~ -
  - Teilweise gelöst (außer Pflichtfeld-Validierung)
- Pflichtfeld-Eigenschaft ist kein Validator
  - offen
- ~~Validierungsregeln pro View definiert~~
  - Teilweise gelöst (außer Pflichtfeld-Validierung)
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - Offen (wäre mit einem Hibernate Backend mit Seam gelöst)

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Fazit: JSF + SEAM + Erweiterung Pflichtfeld

- ~~Vermischung von Anzeige und Validierung~~
  - Vollständig gelöst
- ~~Pflichtfeld-Eigenschaft ist kein Validator~~
  - Vollständig gelöst
- ~~Validierungsregeln pro View definiert~~
  - Vollständig gelöst
- Nur serverseitige Validierung
  - offen
- Validierungen können vom Backend abweichen
  - offen
- Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden
  - Offen (wäre mit einem Hibernate Backend mit Seam gelöst)

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Fazit: JSF + SEAM + Erweiterung Pflichtfeld + WSDL-Validierung

- ~~Vermischung von Anzeige und Validierung~~
  - Vollständig gelöst
- ~~Pflichtfeld-Eigenschaft ist kein Validator~~
  - Vollständig gelöst
- ~~Validierungsregeln pro View definiert~~
  - Vollständig gelöst
- Nur serverseitige Validierung
  - offen
- ~~Validierungen können vom Backend abweichen~~
  - Vollständig gelöst (für Einzelfeldvalidierungen)
- ~~Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden~~
  - Vollständig gelöst

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Fazit: JSF + SEAM + Erweiterung Pflichtfeld + WSDL-Validierung + AJAX

- ~~Vermischung von Anzeige und Validierung~~
  - Vollständig gelöst
- ~~Pflichtfeld-Eigenschaft ist kein Validator~~
  - Vollständig gelöst
- ~~Validierungsregeln pro View definiert~~
  - Vollständig gelöst
- ~~Nur serverseitige Validierung~~
  - Ersetzt durch AJAX (dann schon Validierung bei der Eingabe)
- ~~Validierungen können vom Backend abweichen~~
  - Vollständig gelöst (für Einzelfeldvalidierungen)
- ~~Änderungen in den Restriktionen im Backend müssen an Frontend-Entwickler kommuniziert werden~~
  - Vollständig gelöst

IKANO BANK

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

Gesammelte Erfahrungen

- JSF hatte zunächst nicht alles, was ich erwartet hatte
- Seam bringt viele Funktionen mit, die das Leben mit JSF vereinfachen
- Weitere fehlende Funktionen lassen sich nachrüsten
- Komponentenarchitektur von JSF braucht mehr Rechenzeit und mehr RAM als z.B. Struts 1.x
- Die Komponentenarchitektur bietet viele neue Möglichkeiten
  - Nachträgliche Validierungen hinzufügen
  - Phase-Listener können fehlerhafte Felder markieren
  - Komplett dynamische Formulare
  - Basis für ein einfacheres AJAX

Ganz vorne. JSF-Input-Controls validieren Eingaben gemäß WSDL.

IKANO
BANK