Erweiterung von JSP mit TagLibs

von Thomas Pöschmann

JavaServer Pages sind eine wunderbare Sache, denn sie befreien den Entwickler von den Zwängen der Servlets. Er kann nun innerhalb von HTML problemlos Java einbinden. Allerdings ruft eine mit Scriptlets gespickte JSP-Seite oft das blanke Grauen in den Augen der Leute hervor, welche JSP jeden Tag einsetzen, um die Firmen-Website weiterzuentwickeln und zu warten.

Diese eben genannte Spezies als Vermittler zwischen Entwicklern und Designern möchte in der Regel mit Java nicht viel am Hut haben, obwohl gewisse Grundkenntnisse vorhanden sind. Und letzten Endes leidet auch die Wartbarkeit der Seite extrem unter den zwischen HTML-Tags versteckten Schleifen, Bedingungen und Datenbankabfragen. Selbst eine gute Einrücktaktik löst das Problem nicht, dass nach einer Schätzung von mir 95 % einer Site mit JSP generischer Code ist, der problemlos abstrahiert werden kann.

Warum also nicht gleich diesen Kram hinter Tags verstecken? JSP macht das vor; mit

```
<% Integer i = new Integer(); %>
```

sieht folgender Ausdruck schon viel eleganter aus:

```
<jsp:useBean id="i" class="java.lang.Integer"/>
```

– und benötigt nur noch einen HTML-Editor, der dieses Tag versteht.

Das zweite Beispiel schaut schon nach XML aus und kann durch einen Parser gejagt werden.

Zauberei

Der Trick dabei ist, dass die jsp:-Tags wieder Java-Klassen sind. Nehmen Sie zum Beispiel das obige *useBean*-Tag (XML-Notation: Element), dann fällt auf, dass dieses zunächst einen Namen hat. Außerdem verfügt es über viele Attribute, von denen einige (nämlich die zwei angegebenen) unbedingt vorhanden sein müssen. Schlussendlich kann es weitere Elemente enthalten. Da hier das schließende „>“ allerdings mit einem Slash versehen ist, weisen wir aber darauf hin, dass es keinen so genannten Body-Bereich gibt.

Eine Java-Klasse, welche hinter diesem Tag steht, reagiert nun auf die verschiedenen Bestandteile des Tags. Die JSP-Engine wird uns nämlich mitteilen, wie das Tag aussieht:

- Es gibt ein Attribut *id*, Wert ist *i*.
- Es gibt ein Attribut *class*, Wert ist *Integer*.
- Es gibt keine weiteren (nested) Tags, der Body ist also leer.

Auf diese Beschreibung werden wir als Entwickler des Tags nun reagieren. Die JSP-Engine gibt uns außerdem noch einen Stream mit, welcher direkt auf der Position steht, an welcher das Tag in HTML erscheint – und damit wird das Tag durch uns quasi ersetzt. Dies könnte nun wieder komplexer HTML-Code oder nur ein einfacher Wert (etwa für eine Tabelle) sein. Darüber hinaus können wir Variablen deklarieren und initialisieren, die von anderen Tags genutzt werden.

Nach Erzeugen dieser Klasse, welche die Funktionalität kapselt, die beim Auftreten des Tag aufgerufen wird (Tag-Handler), sind noch weitere Schritte nötig, um ein erstes Beispiel zu entwickeln. Zunächst muss das Tag mit Hilfe eines XML-Deskriptors beschrieben werden. Dort steht etwa der zukünftige Name des Tags, seine Attribute und welche davon „mandantory“ sind, also angegeben werden müssen. Mehrere dieser Beschreibungen ergeben dann eine Bibliothek, auch TagLib genannt. Diese wird zur Nutzung später in eine JSP-Seite eingebunden.

Programmierung

Ein Tag-Handler implementiert die Schnittstelle *javax.servlet.jsp.tagext.Tag* oder die davon erbenden Schnittstellen *IteratorTag* und dem davon erbenden *BodyTag*. Letztere werden für den Anfang aber nicht benötigt.

Zusätzlich werden set-Methoden für die Attribute des Tags implementiert. Deren Übergabeparameter kann immer String sein, jedoch ist ein Mapping zwischen den wichtigsten Java-Datentypen (*boolean*, *byte*, *char*, *double* usw.) definiert, so dass gleich typsicher gearbeitet werden kann.

Tritt nun das entsprechend zugeordnete Tag auf der Seite auf, erzeugt die JSP-Engine möglicherweise eine neue Instanz des Tag-Handlers unter Verwendung seines Leerkonstruktors. Anschließend werden die Attribute, das übergeordnete Tag (Nutzung: siehe später) sowie der PageContext gesetzt. Der *PageContext* ist vom Typ *javax.servlet.jsp.PageContext* und ermöglicht etwa die Arbeit mit Cookies oder Attributen in den Bereichen *Page*, *Request*, *Session* und *Applikation* (siehe später).

Zuletzt erfolgt der Aufruf der Methode *doStartTag()*. Dabei wurde der Inhalt (Body), also etwa Text oder weitere Tags, noch nicht geparst. Hier kann der Tag-Handler nun entscheiden, ob der Body weiter ausgewertet wird oder nicht. Auch das Auslösen einer Exception (*javax.servlet.jsp.JspTagException*) ist möglich. Details dazu finden Sie in Tabelle 1. Sollte während dieser Methode keine Exception auftreten, wird die JSP-Engine den Inhalt des Bodys je nach Rückgabewert parsen und anschließend *doEndTag()* aufrufen. Diesen Prozess sehen sie noch einmal in der Übersicht in Abbildung 1 dargestellt, wobei der Prozess der Konstruktion und Destruktion spekulativ ist, denn Instanzen können nachgenutzt werden.

Das fertige implementierte Tag ist im Quelltext 1 abgedruckt (s. Kapitelende).

Wert	Bedeutung
Tag.SKIP_BODY	Der Inhalt des Bodys soll nicht ausgewertet werden (nur für doStartTag and do AfterBody).
Tag.SKIP_PAGE	Die Seite soll nicht weiter interpretiert werden (nur für doEndTag).
Tag.EVAL_BODY_INCLUDE	Der Inhalt des Bodys soll ausgewertet werden (nur für doStartTag).
Tag.EVAL_PAGE	Die Seite soll weiter ausgewertet werden (nur für doEndTag)
javax.servlet.jsp.JspTag-Exception	Exception, welche die weitere Auswertung der Seite unmöglich macht, und die JSP-Engine anweist, die Fehlerseite anzuzeigen.
IterationTag.EVAL_BODY _AGAIN	Der Inhalt des Bodys soll erneut angezeigt werden (doAfterBody).
BodyTag.EVAL_BODY _BUFFERED	BodyContent wird übergeben (doStart-Tag).

Tabelle 1: Mögliche Rückgabewerte der doXXX-Methoden eines Tag-Handlers

Damit ist allerdings erst die halbe Arbeit erledigt, denn es muss eine Beschreibungsdatei erzeugt werden. Dieser so genannte TagLib Descriptor sieht etwa so aus:

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

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>simple</short-name>
  <uri></uri>

  <tag>
    <name>simple</name>
```

```
    <tag-class>taglib.SimpleTag</tag-class>
    <attribute>
       <name>someAttribute</name>
       <required>true</required>
       <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

</taglib>
```

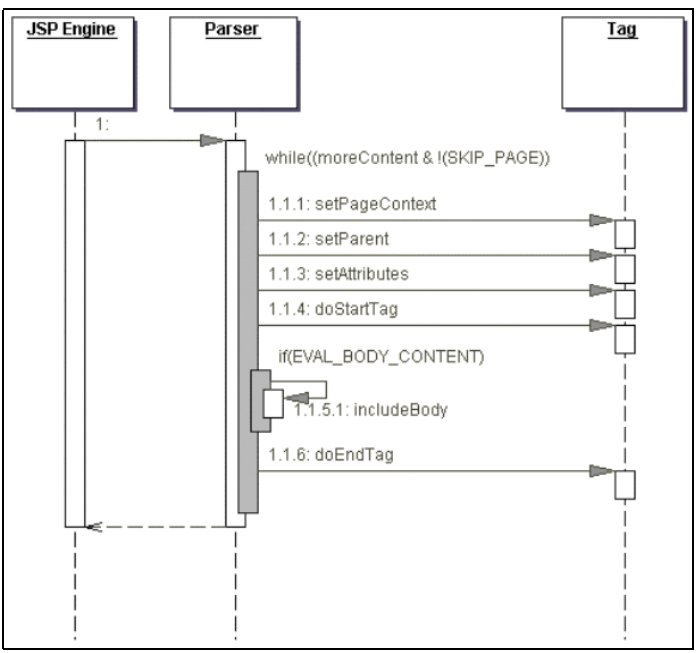

Abbildung 1: Aufruffolge bei Tag

Wichtig ist hier die verwendete JSP-Version, da sich zwischen Version 1.1 und 1.2 die Namen der Elemente geändert haben. Der Name des Tags ist hinter „name“ verborgen. Außerdem werden alle Attribute des Tags kenntlich gemacht, zusammen mit der Information, ob Sie unbedingt gesetzt werden müssen (*required*). Optional ist die Angabe eines Datentyps, dabei übernimmt JSP gleich die Konvertierung (Beispiel siehe

später). Wichtig ist das Element *rtexprvalue*, welches angibt, ob das Attribut selbst wieder aus einem Ausdruck realisiert wird oder nicht. Ist dem so, dann muss der Ausdruck vor der Ausführung noch geparst werden. Ein Beispiel:

```
<sometag att="abc"/>
```

benötigt diese Option nicht, es ist statisch. Beim folgenden Ausdruck dagegen muss der Wert des Attributs noch ausgewertet werden:

```
<sometag att="<%= person.getName() %>"/>
```

Nun muss noch die TagLib in das System eingefügt werden. Dies wird über die Konfigurationsdatei der Web-Applikation (*web.xml*) realisiert:

```
<web-app>
  <welcome-file-list>
    <welcome-file>test.jsp</welcome-file>
  </welcome-file-list>
  <taglib>
    <taglib-uri>http://www.javamagazin.de/taglib</taglib-uri>
    <taglib-location>test.tld</taglib-location>
  </taglib>
</web-app>
```

Die URI hilft dabei, Namenskonflikte zu vermeiden. Es wird hier vorausgesetzt, dass sich die TLD im gleichen Verzeichnis befindet.

Die Einbindung des Tags in eine JSP-Seite geschieht dabei wie folgt, wobei die URI auf ein Präfix aufgelöst wird:

```
<%@ taglib
      uri="http://www.javamagazin.de/taglib"
      prefix="jm" %>

<HTML>
    <BODY>
        <jm:simple someAttribute="Hello, World!"/> <br>
    </BODY>
</HTML>
```

Ist das Attribut nicht vorhanden, zeigt die JSP-Engine entsprechend einen Fehler an, der zur Ausgabe des stacktrace oder zur Weiterleitung auf eine Fehlerseite resultiert.

Fehlerbehandlung

Innerhalb von *doStartTag()* oder *doEndTag()* ausgelöste Exceptions führen in der Regel dazu, dass die JSP-Engine eine Fehlerseite anzeigt. In JSP 1.2 wurde neu eingeführt, dass ein Tag-Handler zusätzlich das Interface *javax.servlet.jsp.tagext.TryCatchFinally* implementieren kann. Hier sehen Sie eine mögliche Aufruffolge der JSP-Engine, sollte der Handler diese Schnittstelle implementieren:

```
tagHandler.setPageContext(pc);
tagHandler.setParent(...);
// setter methods follow
try
{
  doStartTag();
  //...
  doEndTag();
} catch (Throwable t)
{
  //...
  (TryCatchFinally)tagHandler).doCatch(t);
} finally {
  (TryCatchFinally)tagHandler).doFinally();
}
```

Damit kann der Tag-Handler innerhalb von *doCatch()* neue Exceptions erzeugen, die dann zur so genannten Exception-Chain hinzugefügt werden. Bei der Exception-Chain handelt es sich um die angelaufenen Exceptions von der Verursacher-Exception, welche vielleicht durch mehrere catch-Blöcke lief und dann durch andere Exceptions noch einmal gewrappt wurde. Wird Sie nicht erneut in *doCatch()* geworfen, gilt Sie als behandelt, und die Umleitung auf die Fehlerseite wird unterdrückt. Quelltext 2 demonstriert diese Vorgehensweise. Im Übrigen wird hier von der Klasse *TagSupport* geerbt, welche schon ein Standardverhalten (etwa für den *PageContext*) mitbringt. Nur die Methoden, welche von diesem Verhalten abweichen sollen, werden von uns implementiert.

Wird das Beispiel korrekt aufgerufen, erfolgt keine Ausgabe in HTML:

```
<jm:tcf someAttribute="ABC"/>
```

Auf der Kommanozeile bzw. dem Log-File des Servers werden folgende Ausgaben gemacht:

```
taglib.TCFTag.doStartTag()
taglib.TCFTag.doEndTag()
taglib.TCFTag.doFinally()
```

Wird dagegen ein Fehler produziert, ist Folgendes in HTML bzw. im Log-File zu sehen:

```
<jm:tcf someAttribute=""/>
```

```
taglib.TCFTag.doCatch(): here comes the stack trace...
javax.servlet.jsp.JspException: SomeError
...
...TCFTag: end of stack trace <br>
```

```
taglib.TCFTag.doStartTag()
taglib.TCFTag.doFinally()
```

Wird nun die Exception aus *doCatch()* erneut geworfen, leitet die JSP-Engine zur Fehlerseite um.

Voller Zugang

Wünschen Sie Zugriff auf die Informationen des Bodys selbst, weil Sie dessen Inhalt selbst parsen oder verändern wollen, müssen Sie *BodyTag* implementieren oder von *BodyTagSupport* erben. Dann wird Ihnen die JSP-Engine mithilfe der Methode *setBodyContent* ein von *javax.servlet.jsp.JspWriter* erbendes Objekt übergeben. Voraussetzung ist, dass Sie innerhalb von *doStartTag* die Konstante EVAL_BODY_BUFFERED zurückgeben.

Nach dem Übergeben des BodyContent können Sie innerhalb von *doAfterBody()* den Body-Inhalt manipulieren. Zuvor wird die JSP-Engine

einmalig *doInitBody()* (eine Methode ohne Übergabeparameter) aufrufen (siehe Abb. 3).

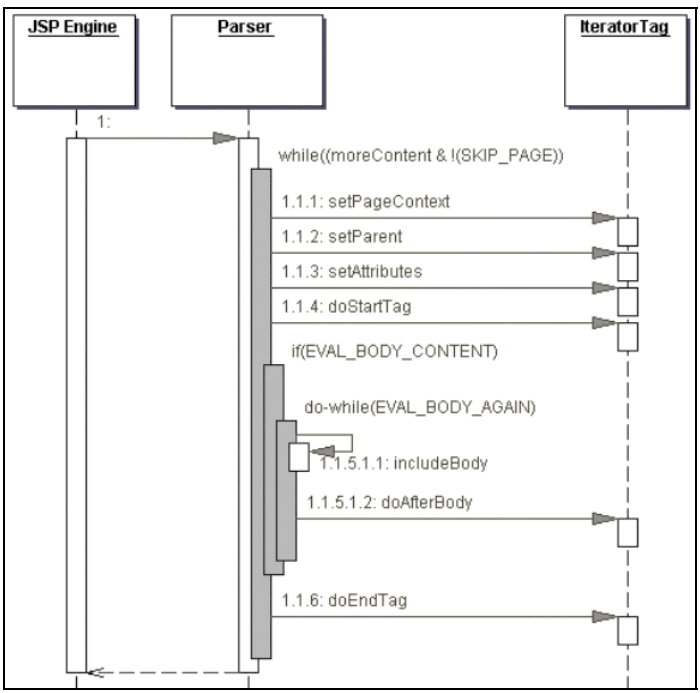

Abbildung 2: Aufruffolge bei IterationTag

Das Beispiel (Quelltext 3) dient zum Suchen & Ersetzen von Zeichen innerhalb des Bodys. Zwar ist dieser Anwendungsfall zunächst trivial; richtig implementiert und angewendet kann er jedoch Cascading Style Sheets (CSS) ablösen. Außerdem fügt das Beispiel jeder Zeile eine Zeilennummer hinzu.

Iterativ

Wie sie bereits sehen konnten, ermöglich das einfache Tag-Interface die Auswertung des Bodys durch die JSP-Engine. Auch die Schachtelung

von Tags ist möglich. Leider kann der so genannte Body nur ein einziges Mal ausgewertet werden.

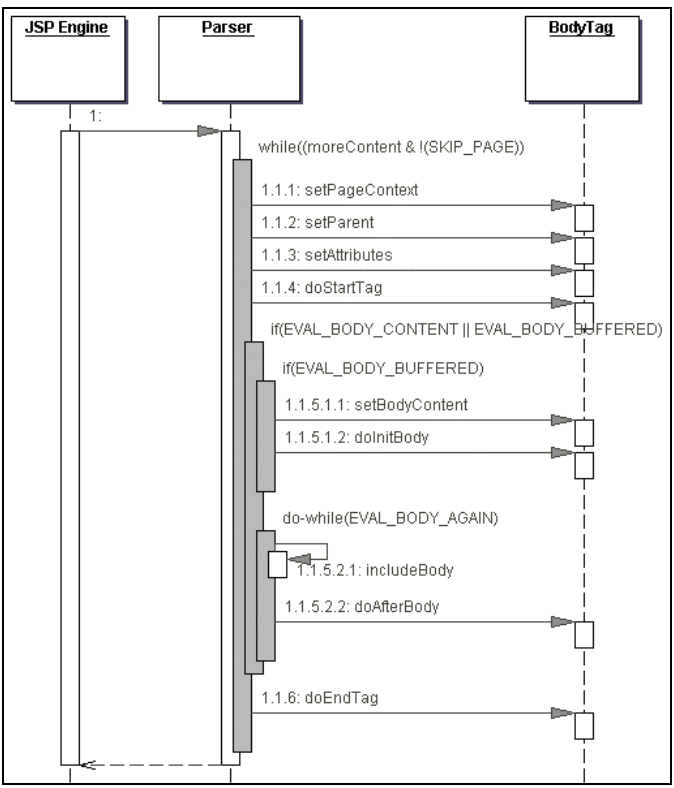

Abbildung 3: Aufruffolge bei BodyTag

Für bestimmte Anwendungsfälle, etwa Schleifen, wäre es schön, den Inhalt des Bodys mehrmals parsen (und damit anzeigen) zu lassen. Bis JSP 1.2 war dies nur mit Hilfe eines Tricks möglich, ab JSP 1.2 wird dieser Prozess durch das so genannte *IterationTag* vereinfacht. Dieses definiert zusätzlich die Methode *doAfterBody()*, welche mit Hilfe der Konstante EVAL_BODY_AGAIN die gewünschte Funktionalität erzielt (siehe Abb. 2). Quelltext 4 verdeutlicht diese Vorgehensweise.

Anstatt die Schnittstelle *IteratorTag* selbst zu implementieren, empfiehlt es sich, von Klasse *javax.servlet.jsp.tagext.TagSupport* zu erben. Diese bietet unter anderem Unterstützung für JSP-IDs (ab JSP 1.2) an. Sehr schön ist außerdem die Methode *findAnchestorWithClass()*. Innerhalb verschachtelter Tags kann man nur mit der Methode *getParent()* auf übergeordnete Tags Bezug nehmen. Sucht man aber ein bestimmtes Tag, etwa um über eine seiner Methoden auf den Zustand zuzugreifen, kann dies schon recht schwierig sein. Die statische Methode *findAnchestorWithClass()* kapselt diesen Algorithmus bereits. Angewendet wird er etwa in einem IF-THEN-ELSE-Tag, wie im Quelltext 5, 6 und 7 zu sehen.

Variablen

Innerhalb von JSP-Seiten und Servlets kann jedem Nutzer ein eindeutiger Raum für private Objekte zugewiesen werden. Dieser so genannte Session-Context könnte etwa von einem Servlet dazu genutzt werden, die Kreditkartennummer eines Nutzers zu speichern, um Sie später in einer JSP-Seite wieder auszugeben. Dieser etwas scherzhaft gemeinte Satz hat allerdings einen wichtigen Hintergrund. Der Session-Context eines Nutzers ist als eine Art *Map* zu betrachten, welche String als Schlüssel verwendet und beliebige Objektreferenzen erlaubt. Der Zugriff auf den Session-Context wird mit folgenden Methoden realisiert:

```
Servlet:
  HttpSession session = request.getSession();
  session.setAttribute();
  session.getAttribute();
JSP Seite:
  pageContext.setAttribute();
  pageContext.getAttribute();
```

Da ein HTTP-Request einen Nutzer nicht eindeutig identifiziert, werden in der Regel Cookies oder in URL eingearbeitete Parameter genutzt, um den Nutzer eindeutig zu identifizieren. Der Nachteil des Cookies ist natürlich, dass er vom Nutzer deaktiviert werden kann. Nachteil des URL-Parameters (URL rewriting) ist, dass wirklich alle Links (GET/POST) diesen Parameter enthalten müssen. Bereits eine Lücke, also ein unacht-

sam gesendetes Formular oder ein nicht encoded Link unterbrechen die Kette und verhindern, das der Nutzer eindeutig identifiziert werden kann.

Die eine Möglichkeit, mit anderen Tags in der JSP-Seite zu kommunizieren, besteht aus dem genannten Session-Context. Daneben existieren noch weitere Gültigkeitsbereiche: *request* (für die aktuelle Anfrage, auch wenn mit Hilfe von *including* andere Seiten eingebunden werden), *page* (tatsächlich nur die aktuelle Seite, egal ob Ausgaben anderer Seiten eingebunden werden) und *application* (global, ohne an Nutzer, Seiten oder Requests gebunden zu sein.

Diese Varianten haben den Nachteil, dass andere Tags (oder gar Scriptlets) erst Methoden aufrufen müssen, um an die Referenzen zu gelangen. Daher bietet JSP die Möglichkeit, Variablen (also benannte Objektreferenzen) anzulegen, welche dann sofort in der Seite genutzt werden können.

Quelltext 8 zeigt ein Tag, in welchem der Nutzer von außen den Namen einer Variablen der JSP-Seite setzt (*variableName*). Außerdem wird ein Wert für diese Variablen (*value*) gesetzt. Die Nutzung dieses Tag könnte wie folgt aussehen:

```
<jm:declare variableName="myvar" value="TheValue">
  <%= myvar %>
</jm:declare>
```

ergibt

```
TheValue
```

Wenn Sie allerdings Quelltext 8 anschauen, wird nirgendwo eine Variable deklariert – lediglich ein String wird im PageContext hinterlegt:

```
    pageContext.setAttribute(variableName, value);
```

Die Lösung dieses Rätsels besteht in einer zusätzlichen Klasse, der so genannten *TagExtraInfo* (Quelltext 9). Diese sagt aus, dass das Tag eine neue Variable deklariert. Der Name der Variablen wird zur Laufzeit aus

dem Attribut *variableName* des Tag gebildet, der Typ ist String. Die Variable ist nur innerhalb des Tag-Bodys gültig, andere Optionen machen Sie etwa für die ganze Seite ab Auftreten des Tags gültig.

Den Namen der *TagExtraInfo*-Klasse teilen Sie der Engine über den TagLib Descriptor mit:

```
<tag>
  <name>declare</name>
  <tag-class>taglib.DeclareVariable</tag-class>
  <tei-class>taglib.DeclareVariableTEI</tei-class>
  <attribute>
    <name>variableName</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>value</name>
    <required>true</required>
  </attribute>
</tag>
```

Und weiter?

Diese Beispiele sollten zeigen, wie einfach es ist, eigene Funktionen in JSP-Tags zu kapseln. Im Struts-Kapitel werden Sie häufig mit neuen Tags in Berührung kommen. Sollten Sie eine Funktionalität in Struts vermissen, können Sie diese nun leicht selbst in einem Tag hinterlegen.

Literatur

[1] JavaServer Pages Specification, Version 1.2

[2] Tab Libraries Tutorial (Sun)
http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html

[3] Jakarta TabLibs Tutorial (Apache)
http://jakarta.apache.org/taglibs/tutorial.html

Quelltexte

Quelltext 1: SimpleTag.java

```
package taglib;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class SimpleTag implements Tag
{

  private PageContext ctx;
  private Tag parent;
  private String someAttribute;

  public void setPageContext(PageContext parm1)
  {
    ctx = parm1;
  }

  public void setParent(Tag parm1)
  {
    parent = parm1;
  }

  public Tag getParent()
  {
    return parent;
  }

  public int doStartTag() throws JspException
  {
    try
    {
      JspWriter out = ctx.getOut();
      out.print("Hello, your attribute is " + someAttribute);
    } catch (IOException ex)
    {
      throw new JspException(ex);
    }
    return Tag.SKIP_BODY;
  }

  public int doEndTag() throws JspException
  {
    return Tag.EVAL_PAGE;
```

```
  }

  public void release()
  {
  }

  public void setSomeAttribute(String newSomeAttribute)
  {
    someAttribute = newSomeAttribute;
  }


}
```

Quelltext 2: TCFTag.java

```
package taglib;

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

public class TCFTag
  extends TagSupport
  implements TryCatchFinally
{

  public void doCatch(Throwable parm1) throws Throwable
  {
    try
    {
      JspWriter writer = pageContext.getOut();
      writer.println(
        "taglib.TCFTag.doCatch(): stack trace<br>");
      parm1.printStackTrace(new PrintWriter(writer));
      writer.println("...TCFTag: end of stack trace <br>");
      //throw new IOException(
        "Some test that comes through!");
    } catch (IOException ex)
    {
      throw ex;
    }
  }

  public void doFinally()
  {
    System.out.println("taglib.TCFTag.doFinally()");
  }

  public int doStartTag() throws JspException
```

```
  {
    System.out.println("taglib.TCFTag.doStartTag()");
    if (
      (someAttribute == null) ||
      (someAttribute.equals("")) )
      throw new JspException("SomeError");
    return Tag.SKIP_BODY;
  }

  public int doEndTag() throws JspException
  {
    System.out.println("taglib.TCFTag.doEndTag()");
    return Tag.EVAL_PAGE;
  }
  public void setSomeAttribute(String newSomeAttribute)
  {
    someAttribute = newSomeAttribute;
  }

  private String someAttribute;

}
```

Quelltext 3: SearchAndReplace.java

```
package taglib;

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

public class SearchAndReplace extends BodyTagSupport
{

  public int doStartTag() throws JspException
  {
    return BodyTag.EVAL_BODY_BUFFERED;
  }

  public int doAfterBody() throws JspException
  {
    return Tag.SKIP_BODY;
  }

  public int doEndTag() throws JspException
  {
```

```
    try
    {
      BufferedReader in =
        new BufferedReader(bodyContent.getReader());
      bodyContent.clearBody();
      String line;
      int lineNumber = 0;
      while (( line = in.readLine() ) != null)
      {
        if (line.length() > 0)
        {
          line = line.replace(fromChar, toChar);
          bodyContent.getEnclosingWriter().println(
            ++lineNumber + " " + line);
        }
      }
    } catch (Exception ex)
    {
      throw new JspException(ex);
    }
    return Tag.EVAL_PAGE;
  }
  public void setFromChar(char newFromChar)
  {
    fromChar = newFromChar;
  }

  public void setToChar(char newToChar)
  {
    toChar = newToChar;
  }

  public void setBodyContent(BodyContent parm1)
  {
    System.out.println("Body content set");
    this.bodyContent = parm1;
  }

  private char fromChar;
  private char toChar;
  private BodyContent bodyContent;

}
```

Quelltext 4: RepeatTag.java

```
package taglib;

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

public class RepeatTag
  extends TagSupport
  implements IterationTag
{

  public int doStartTag() throws JspException
  {
    return EVAL_BODY_INCLUDE;
  }

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

  public int doAfterBody() throws JspException
  {
    return
    ( ++count <repeat ? EVAL_BODY_AGAIN : EVAL_BODY_INCLUDE);
  }

  public void setRepeat(Integer newRepeat)
  {
    repeat = newRepeat.intValue();
  }

  private int repeat = 0;
  private int count = 0;


}
```

Quelltext 5: If.java

```
package taglib;

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

public class If extends TagSupport
{

  public int doStartTag() throws JspException
  {
    return Tag.EVAL_BODY_INCLUDE;
  }

  public void setCondition(boolean newCondition)
  {
    thenCondition = new Boolean(newCondition);
    elseCondition = new Boolean(!newCondition);
  }

  public boolean isThenCondition() throws JspException
  {
    if (thenCondition == null)
      throw new JspException(
        "Only one THEN tag allowed inside an IF tag!");

    boolean result = thenCondition.booleanValue();
    thenCondition = null;
    return result;
  }

  public boolean isElseCondition() throws JspException
  {
    if (elseCondition == null)
      throw new JspException(
        "Only one ELSE tag allowed inside an IF tag!");

    boolean result = elseCondition.booleanValue();
    elseCondition = null;
    return result;
  }

  private Boolean thenCondition;
```

```
  private Boolean elseCondition;


}
```

Quelltext 6: Then.java

```
package taglib;

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

public class Then extends TagSupport
{

  public int doStartTag() throws JspException
  {
    If parentIf = (If) findAncestorWithClass(this, If.class);

    if (parentIf == null)
      throw new JspException(
        "THEN or ELSE tags have to be inside an IF tag!");

    if (parentIf.isThenCondition())
      return Tag.EVAL_BODY_INCLUDE;
    else
      return Tag.SKIP_BODY;
  }
}
```

Quelltext 7: Else.java

```
package taglib;

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

public class Else extends TagSupport
{

  public int doStartTag() throws JspException
  {
    If parentIf = (If) findAncestorWithClass(this, If.class);
```

```
    if (parentIf == null)
      throw new JspException(
        "THEN or ELSE tags have to be inside an IF tag!");

    if (parentIf.isElseCondition())
      return Tag.EVAL_BODY_INCLUDE;
    else
      return Tag.SKIP_BODY;
  }
}
```

Quelltext 8: DeclareVariable.java

```
package taglib;

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

public class DeclareVariable extends TagSupport
{

  public int doStartTag() throws JspException
  {
    pageContext.setAttribute(variableName, value);
    return Tag.EVAL_BODY_INCLUDE;
  }

  public void setVariableName(String newVariableName)
  {
    variableName = newVariableName;
  }
  public String getVariableName()
  {
    return variableName;
  }

  public void setValue(String newValue)
  {
    value = newValue;
  }
  public String getValue()
  {
    return value;
```

```
  }

  private String variableName;
  private String value;

}
```

Quelltext 9: DeclareVariableTEI.java

```
package taglib;

import javax.servlet.jsp.tagext.*;

public class DeclareVariableTEI extends TagExtraInfo
{

  public VariableInfo[] getVariableInfo(TagData data)
  {
    return new VariableInfo[]
    {
      new VariableInfo(
        data.getAttributeString("variableName"),
        "java.lang.String",
        true,
        VariableInfo.NESTED
      )
    };
  }


}
```