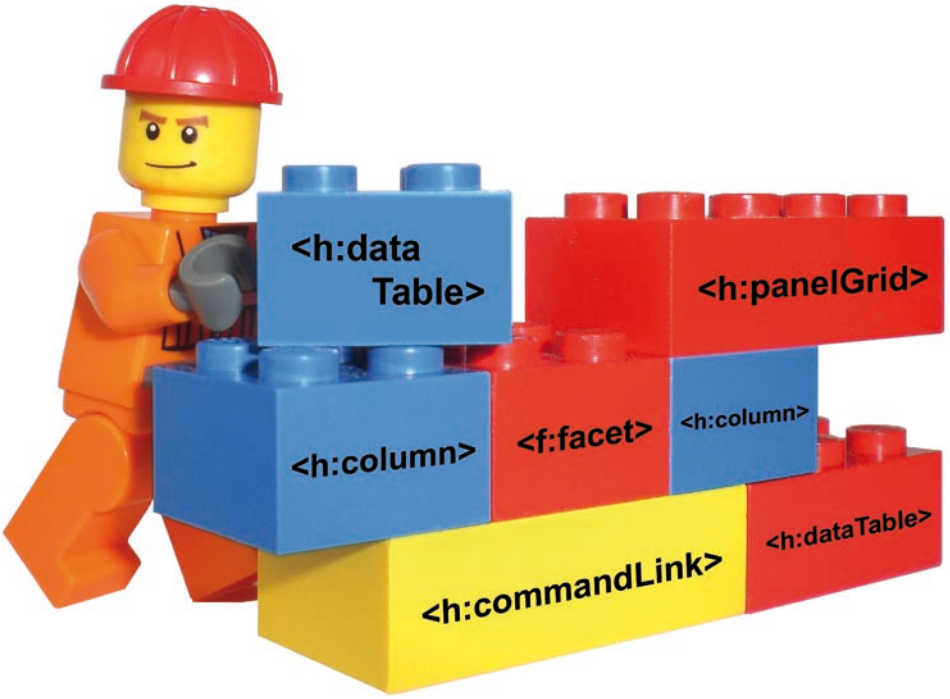

Lego **oder** Playmobil?

Bausteine zusammenzusetzen macht Spaß. Man kombiniert die Komponenten, die Anzahl an Basiskomponenten ist überschaubar und die Möglichkeiten, diese zu kombinieren, sind unbegrenzt. Einen halbwegs vollständigen Überblick über die Patterns und Antipatterns eines Baukastens gewinnt aber nur derjenige, der schon Projekte damit umgesetzt hat.

von Ganesh Jung

Wer schon einmal mit einem 4-jährigen Kind Lego gebaut hat, hat erlebt, wie das 4-Jährige beim Aufbau von Mauern immer hübsch einen Stein auf den andern setzt, anstatt die Steine gegeneinander zu versetzen. Später wundert es sich, dass die lose nebeneinander stehenden Steintürmchen immer umfallen. Wer einem JSF-Entwickler mit 3 Monaten Entwicklungserfahrung die Aufgabe stellt, aus einer Tabelle heraus für jede Zeile eine Detailsicht anzuzeigen, erlebt Ähnliches. Der Entwickler schachtelt in eine *h:column* seiner *h:dataTable* einen *h:commandLink*, der eine Methode seiner Backing Bean triggert, und wundert sich, dass nun die *h:commandLink* Tags aller Zeilen dieselbe Methode triggern, und beim Ausführen der Methode nicht mehr bekannt ist, welche Zeile die Methode getriggert hat.

Den Quellcode zu diesem Artikel finden Sie unter www.javamagazin.de

Probleme durch Komponentenbibliotheken

Wenn das Kind und der Entwickler Glück haben, ist nun Papa (bzw. ein Coach) in der Nähe und verrät die Lösung (manchmal reicht auch ein Artikel im Java Magazin). Ansonsten kommt eventuell Frust auf und der Ruf nach einem „besser funktionierenden" Bausatz wird laut. Das Kind wird sich seinem Playmobil-Bauernhof zuwenden, wo die Wände auf eindeutige Weise verbunden sind und 2 oder 3 Möglichkeiten existieren, den Bauernhof umzubauen. Der Entwickler sucht im Web nach einer Komponentenbibliothek, die sein Problem löst. Er findet Lösungen in den Livedemos von Tomahawk, Trinidad, Tobago, RichFaces oder IceFaces.

Was beide (Kind und Entwickler) jetzt verlieren, ist zu diesen Zeitpunkt vielleicht gar nicht bewusst: Sie verlassen den Standard. Die ersten Erfolge sind umwerfend: Da steht so eine Hauswand mit einem Handgriff und ist sofort mit

dem nächsten Teil verbunden. Schwierigkeiten kommen auf, wenn das Fenster ein paar Zentimeter versetzt werden soll. Mit der JSF-Komponentenbibliothek kommen weitere Aspekte hinzu:

Wer Standardkomponenten verwendet, darf erwarten, dass Upgrades des JSF-Standards entweder abwärtskompatibel vorgenommen werden oder Migrationspattern angeboten werden. Im schlimmsten Fall stellt man im Jahre 2010 fest, dass die verwendete Komponentenbibliothek nicht mehr weiterentwickelt wird und ein Upgrade auf eine neue JSF-Version somit eine Umstellung der gesamten Anwendung bedeutet.

Wer die Standardkomponenten verwendet, darf außerdem erwarten, dass seine Anwendung mit jeder JSF-Implementierung funktioniert. Die Umstellung von z.B. Glassfish auf BEA10 ist so nur eine Frage der Konfiguration und wird dadurch garantiert, dass der Application Server SUNs Kompatibilitätstest bestanden hat. Ob eine Komponentenbibliothek die Umstellung klaglos mitmacht, hängt von der Implementierung ab. Da möglicherweise der Application Server erst nach der Komponentenbibliothek entwickelt wurde, kann auch keine Komponentenbibliothek die Lauffähigkeit auf allen JSF-kompatiblen Application Servern garantieren.

Erweiterungen der Standardbibliothek wie die Integration in einen Portlet Server werden nur von manchen Bibliotheken unterstützt. Die Frage, ob eine solche Erweiterung eingesetzt werden soll, kann somit davon abhängig sein, welche Komponentenbibliotheken bereits eng mit der Anwendung verflochten sind.

Die Verbreitung einer JSF-Komponentenbibliothek muss immer hinter der Fehlerrate der Standardkomponenten zurückbleiben, weil die Standardkomponenten von einem größeren Nutzerkreis verwendet werden. Folge ist, dass die Fehlerrate der Komponentenbibliothek durch den geringeren Grad an Praxistests höher ausfällt.

Ein Problem, das mit Standardkomponenten lösbar ist, aus Unkenntnis des richtigen Implementierungspatterns mit einer Erweiterungskomponente zu lösen, führt also zu erschwerten Upgrades, einer erschwerten Migration, schlechterer Erweiterbarkeit und einer größeren Wahrscheinlichkeit, sich mit Bugs in den Komponenten herumschlagen zu müssen [1].

„Wiederwegwerfbarkeit“

Die Probleme, die durch Komponentenbibliotheken geschaffen werden, aufzuzählen, heißt nicht, diese abzulehnen: Erweiterungskomponenten für JSF sind nützlich und gewollt. Der Komponentengedanke ist Kern der JSF-Philosophie. Um Erweiterbarkeit und Migrierbarkeit zu erhalten ist es allerdings hilfreich, den Standard überall dort zu verwenden, wo es möglich ist, und nur dort Komponenten hinzuzunehmen, wo der Standard nicht ausreicht. Eine „gute“ Komponente sollte daher den Standard erweitern, ohne dabei der Anwendung ein neues Pattern aufzuzwingen. Eine Komponente auszutauschen oder zu entfernen, soll ohne große Anpassungen des restlichen Codes möglich sein. Im Idealfall entwickelt man die Anwendung zunächst auf Basis des Standards und fügt Erweiterungen erst zuletzt hinzu. Durch Einsatz der Erweiterungen wird die Anwendung leichter zu bedienen und die Benutzererfahrung flüssiger, die Komponenten können aber leicht wieder entfernt werden. Diese Eigenschaft einer Komponente nenne ich „Wiederwegwerfbarkeit“. Die Wiederwegwerfbarkeit ist die große Schwester der Wiederverwendbarkeit. Auf Englisch wird oft der Begriff „non intrusiveness“ verwendet, um auszudrücken, dass eine wiederverwendbare Komponente den Verwender nicht abhängig macht, also wiederwegwerfbar ist. Die Wiederwegwerfbarkeit ist zum einen abhängig von der Bauart der Komponente und zum anderen hängt sie auch von der Art ab, wie die Komponente eingesetzt wird. Wiederwegwerfbarkeit erzeugt man also sowohl bei der Auswahl als auch bei der Verwendung einer Komponente.

Implementierungs-Pattern für Standard-JSF-Komponenten

Für ein paar Eingabefelder und Submit-Knöpfe braucht man kein neues Framework. Eine einfache JSP tut es für diesen Zweck auch. Interessant wird die Verwendung eines Frameworks, wenn komplexere Oberflächenelemente dargestellt werden sollen. Ein erstes Urteil über ein Framework bilde ich mir gerne, indem ich mir anschaue, wie die Darstellung von Tabellen gelöst ist. Wie einfach ist es, eine einfache Liste vom Objektmodell auf die Oberfläche zu zaubern? Wie flexibel ist die Darstellung der Oberfläche? Und was muss der Entwickler einbauen, damit Eingaben aus den Tabellenzeilen wieder in das Modell zurückgeschrieben werden?

JSF bietet überzeugende Antworten auf diese Fragen. Die meisten Probleme bei der Entwicklung von Webseiten lassen sich bei der Verwendung von Java Server Faces auf intuitive Art lösen. Es gibt aber bei der Verwendung von JSF ein paar Probleme, die nur dann leicht mit Standardkomponenten lösbar sind, wenn man das richtige Implementierungspattern kennt. Selbst JSF-Entwicklern mit mehreren Monaten Entwicklungserfahrung sind diese Patterns oft nicht bekannt. Ich habe schon einige

Implementierungshilfen für eine leichte Umsetzung des Stacked Table Patterns (siehe Aufmacherbild)

Dem Baum wird ein *h:panelGrid* für den Header vorangestellt (siehe „External Header“). Die Wurzel des Baums wird als weiteres *h:panelGrid* gerendert. Für die Liste, die unterhalb der Wurzel liegt, wird eine *h:dataTable* verwendet. Die Tabelle wird mit nur einer einzigen Spalte gerendert. Jede Zeile der Liste enthält einen Baumknoten und (optional) eine weitere Liste mit darin geschachtelten Knoten. Die einzige Spalte der Tabelle benötigt dazu ein *h:panelGrid*, das den jeweiligen Baumknoten darstellt, und eine weitere *h:dataTable*-Komponente, die die Liste der geschachtelten Knoten rendert. Für jede Knotenebene des darzustellenden Baums wird so eine Tabelle erstellt, in die die tieferliegenden Knoten geschachtelt werden.

Aha-Effekte erleben dürfen, wenn ich beim Coachen von JSF-Entwicklern diese einfachen Lösungen erklärt habe.

Implementierungs-Pattern 1: Row Method

Dieses Pattern wird verwendet, um in einer *h:dataTable* in jeder Zeile einen Hyperlink zu verwenden und in der verlinkten JSF Action Zugriff auf die Daten der angeklickten Zeile zu haben.

Tabellen mit dynamischer Zeilenzahl werden in JSF mit dem *h:dataTable*-Element implementiert. Das Datenmodell einer *h:dataTable* besteht aus einer Liste von Zeilenobjekten. Die Tabellenspalten werden durch das *h:column-Objekt* gebildet, in welches Standard-JSF-Bausteine geschachtelt werden, wodurch höchstmögliche Flexibilität gewährleistet wird. Jede Komponente, die außerhalb einer Tabelle eingesetzt werden kann, kann auch in eine Tabellenspalte gesetzt werden.

In vielen Fällen soll nun aus jeder Tabellenzeile heraus eine Aktion für das Zeilenobjekt gestartet werden (z.B. Zeilenauswahl oder Detailansicht). Wie im einleitenden Text beschrieben, gerät der Entwickler leicht auf den Pfad, einen *h:commandLink* in eine *h:column* zu schachteln und eine Methode der Backing Bean als Action anzugeben. Da der *h:commandLink* auf diese Weise aus jeder Tabellenzeile dieselbe Methode triggert und an JSF Actions keine Parameter übergeben werden können, ist bei der Ausführung der Methode nicht mehr bekannt, welche Tabellenzeile geklickt wurde.

Die Problematik besteht also darin, die Methode so aufzurufen, dass sie weiß, welche Tabellenzeile bearbeitet werden soll. Der Trick besteht darin, die getriggerte Methode in das Zeilenobjekt zu legen. Die Action verweist auf die Methode, die im Zeilenobjekt implementiert ist, die Methode kann einfach auf die Properties des Objekts greifen, zu dem sie gehört, und weiß so, welche Zeile geklickt wurde.

Single Row Selection mit Row Method Pattern

Mithilfe des Row Method Patterns kann man beispielsweise eine Tabelle implementieren, die beim Klick auf eine Zeile diese Zeile markiert darstellt (Einzelzeilenauswahl).

In Nicht-JSF-Anwendungen wird hierfür manchmal eine Gruppe von Radiobuttons verwendet. In jeder Zeile befindet sich dann ein Radiobutton, von denen höchstens einer angeklickt ist, weil alle Radiobuttons zu derselben Gruppe gehören. Mit JSF geht das so nicht, denn eine Gruppe von Radiobuttons und eine *h:dataTable* können nicht ineinander verschränkt werden. Später kommen wir zum Script Injection Pattern, das eine Lösung dieses Problems mit JavaScript ermöglicht.

Mit dem Row Method Pattern kann man das Problem folgendermaßen angehen: Jedes Zeilenobjekt enthält eine Methode *select()*, die die Zeilenauswahl ansteuert, und der Container enthält eine *selectedId*, die die ID der gewählten Zeile enthält.

Das Beispiel lässt sich auch so abwandeln, dass nicht eine Zeile, sondern mehrere Zeilen pro Tabelle auswählbar werden. Dazu wird in jedes Zeilenobjekt eine Eigenschaft

Überblick über die Pattern

- *Implementierungs-Pattern 1: Row Method*
  Wenn in einer JSF-Tabellenzeile ein Hyperlink oder ein Button positioniert wird, sollte die getriggerte Action eine Methode des Zeilenobjekts sein. So kann während der Ausführung der Methode einfach auf die weiteren Attribute der angeklickten Tabellenzeile zugegriffen werden. Dieses Pattern ist so einfach, dass ich es gar nicht erwähnen würde, wenn ich nicht bei mir selbst und bei anderen Entwicklern gesehen hätte, dass es ohne Anleitung Monate dauert, ehe man darauf kommt.
- *Implementierungs-Pattern 2: External Header*
  Dieses Pattern wird benötigt, wenn die Header-Zeile einer Tabelle und der Tabellen-Body unterschiedliche Eigenschaften haben sollen. Man legt den Header in eine eigene Tabelle (mit dem *h:panelGrid* Tag) und kann so die CSS Styles des Headers und des Bodys getrennt setzen. So können ein vertikaler Scrollbar für den Body, eine zweizeilige Header-Zeile oder ein anklickbarer Header, der den Body auf- und zuklappt, leicht realisiert werden. Ich empfehle, jede JSF-Tabelle mit External Header zu entwickeln. Zum Entwicklungszeitpunkt kostet das nur wenig Extraarbeit, und wenn später die Anforderungen erweitert werden, ist der Rahmen schon vorhanden, vor allem sind dann keine CSS-Style-Anpassungen nötig, um auf den External Header umzustellen.
- *Implementierungs-Pattern 3: Script Injection*
  Ein JavaScript, das am Ende der JSF-Seite läuft, sucht nach Elementen, deren Namen eine Naming Convention befolgen, und legt JavaScript-Funktionen auf die gewünschten Events der Elemente. Mit dem beiliegenden Injection Script [2] können JSF-Tabellen mit Rollover Effect und einzeln oder mehrfach anklickbaren Zeilen versehen werden. Radiobuttons können über die ganze Seite verteilt und auch in Tabellenzeilen verwendet werden. Ist es nicht toll, wie einfach sich die Standard-JSF-Komponenten durch Script Injection „aufbohren" lassen?
- *Implementierungs-Pattern 4: Stacked Table*
  Mit Standard-JSF-Komponenten lassen sich komfortable Bäume aufbauen. Dazu werden *h:dataTable*-Elemente ineinander verschachtelt. Die *h:dataTable* erhält nur eine Spalte (als *h:column*), der Wurzelknoten wird als External Header realisiert. Die *h:column* besitzt wiederum ein h:panelGrid für den Knoteninhalt und eine *h:dataTable* für die Kindknoten. Je nach benötigter Baumtiefe wird diese Rekursion weiter fortgesetzt. Der Clou: Dieser Baum arbeitet auf einem Modell, das einfach aus einer Liste (*java.util.List*) besteht, deren Listenelemente wiederum Listen enthalten. So macht das Entwickeln von Baumelementen Freude!
- *Implementierungs-Pattern 5: Script Binding*
  JSF-Komponenten können einander mit JavaScript ansprechen, wenn sie über ihr *binding*-Attribut an eine Backing Bean gebunden sind und die Backing Bean die ClientId über einen Getter herausgibt. So können all die kleinen JavaScript-Tricks, die man mit JSPs verwendet hat, auch in JSF zum Einsatz kommen.

```
private boolean selected;
```

gelegt, das durch eine Methode

```
public String toggle() {
 selected = !selected;
 return null;
}
```

umgesetzt wird. Wenn durch einen EL-Ausdruck der Style der selektierten Zeile verändert wird – z.B. durch

```
styleClass="#{item.selected ? 'rowSelected' : '' }"
```

dann ist das Ergebnis komplett und die Tabelle hat anklickbare Zeilen. Der Weg dahin ist ganz einfach, oder?

Implementierungs-Pattern 2: External Header

Die Spaltenüberschriften (Header) einer *h:dataTable* können als *<f:facet name="header">* innerhalb der *h:column* angegeben werden. JSF rendert in HTML dann ein THEAD Tag mit einem TH für jede Spalte. Allerdings können Spaltenanzahl und -breite und die CSS Styles der Spalten nicht getrennt für Header und Body vergeben werden. Weiterhin ist es nicht möglich, dem TBODY einen eigenen Style zu geben. Oft sind aber die Anforderungen an eine Anwendung genau solcher Art, dass Header und Body unterschiedliche CSS Styles benötigen.

Ein Beispiel für die Probleme, die man mit verbundenem Header und Body hat, ist die Implementierung eines vertikalen Scrollbars, der nur die Tabellenzeilen scrollt, den Header aber stehen lässt. Ein Weiteres sind Baumdarstellungen (siehe„ Stacked Table"), die beim Klick auf die Header-Zeile den Body auf- und zuklappen. Ein drittes Beispiel, das die entstehenden Schwierigkeiten aufzeigt, ist die Anforderung, Spaltenüberschriften zu gruppieren und jeder Überschriftengruppe wieder eine eigene Überschrift zu geben. Hierzu werden zwei Überschriftenzeilen benötigt, die eigene Spaltenanzahlen und -breiten haben.

Wie kann der Entwickler es mit JSF-Standardkomponenten schaffen, Tabellen-Header und -Body so zu trennen, dass die Eigenschaften für beide Teile getrennt vergeben werden können? Am besten ist es, für die Überschriftenzeile einer *h:dataTable* ein weiteres *h:panelGrid* vor die Tabelle zu stellen, das die Überschriften rendert. Die Tabelle selbst wird mit nur einer einzigen *h:column* erstellt, die wiederum ein *h:panelGrid* enthält. Wenn man nun dieselben CSS-Style-Klassen für die Spalten der Überschriften „*h:panelGrid*" und „*h:panelGrid*" verwendet, die in die *h:column* geschachtelt sind, dann ergibt sich ganz von selbst, dass Header und Spalte jeweils genau untereinander stehen. Die Spaltenformatierung für Header und Body ist darüber hinaus völlig unabhängig konfigurierbar und der Body kann auch mit einem vertikalen Scrollbar versehen werden.

Styling von External Headers

Um diese Lösung auch grafisch ansprechend zu gestalten, ist es notwendig, die Tabellen durch CSS Styles abstandsfrei untereinander zu stellen und zu verhindern, dass die Breiten der Tabellenspalten aufgrund der Breite des Zelleninhalts von den Vorgaben abweichen. Dazu definiert man für die TABLE-, TR- und TD-Elemente jeweils die Eigenschaften *border-collapse* und *border*, die die Ränder zwischen den Tabellen und das Padding, das die Browser per Default um die Tabellenspalten legt, entfernen. Durch den CSS Style *table-layout:fixed* erreicht man, dass beim Verändern der Größe des Browserfensters die vorgegebenen Höhen und Breiten exakt eingehalten werden. Ohne diese Angabe verwirft der Browser die Vorgaben, wenn manche Werte, die in den Zellen stehen, die vordefinierten Größen überschreiten. Dieser Style vergrößert die Tabelle auf 100 % und legt die Zellenhöhe auf 20 px fest.

Vertikaler Scrollbar mit External Header Pattern

Der Einbau eines vertikalen Scrollbars für den Body benötigt noch 2 Erweiterungen des External Header Patterns. Um den Body, also um das *h:dataTable* Tag herum, wird ein HTML DIV Layer mit CSS Style gelegt:

```
<f:verbatim><div class="tablebody"></f:verbatim>
…Tabelle…
<f:verbatim></div></f:verbatim>
```

Die Styleklasse wird dann so definiert:

```
.tablebody {
 /*gewünschte Höhe des Bodys bitte hier eintragen: */
 height:100px;
 overflow-x:none;
 overflow-y:scroll;
}
```

Die zweite Erweiterung bezieht sich auf die Header-Zeile. Dadurch, dass der Body nun rechts einen Scrollbar hat, ist weniger Platz für die Spalten. Der Header braucht also ganz rechts eine zusätzliche Spalte, die leer bleibt und oberhalb des Scrollbars positioniert ist. Diese Spalte kommt als letzte in das Header *h:panelGrid*:

```
<h:outputText escape="false" value=" " />
```

An die *columnClasses* des Headers wird zusätzlich zu den Spaltenstyles, die auch

Listing 1: Single Row Selection mit Row Method

```
public class ListItems extends ArrayList {
 private Long selectedId;
 public Long getSelectedId() {
  return selectedId
  }
  public void setSelectedId(Long selectedId) {
   this.selectedId= selectedId
  }
}

public class Item {
  private Long id;

  public String toggle() {
   FacesContext context = FacesContext.getCurrentInstance();

   /* Variante für JSF 1.1 */
   ((ListItems)context.getApplication().
                createValueBinding("#{listItems}").getValue(context)).
                                                setSelectedId(id);

   /* Variante für JSF 1.2
   ((ListItems)context.getApplication().getExpressionFactory().
                                                createValueExpression
              (context.getELContext(),"#{listItems}",ListItems.class).
                getValue(context.getELContext())).setSelectedId(id);

   /* null zurückzugeben löst einen Roundtrip aus */
   return null;
  }
}
```

für den Body definiert sind, noch ein CSS Style *scrollCol* angehängt:

```
td.scrollCol {
/* Das ist genau die Breite eines Scrollbars */
 width: 17px;
}
```

Implementierungs-Pattern 3: Script Injection

Das *h:dataTable* Tag nimmt dem Benutzer die Erstellung der üblichen HTML-Tabellen-Tags ab. Die Tags TABLE, TBODY, TR und TD werden automatisch gerendert. Mithilfe der *rowClasses* und *columnClasses* können auch die CSS Styles dieser Tags gesetzt werden. Es ist aber nicht möglich, JavaScript-Attribute wie *onClick*, *onMouseOver* oder *onMouseOut* in die generierten Tabellenzeilen zu setzen. Dies hat zur Folge, dass alle Funktionalitäten, die JavaScript Events auf Zeilenebene benötigen, von Standard-JSF-Komponenten nicht direkt unterstützt werden. Hierzu gehören die Einfach- und Mehrfachauswahl von Zeilen – für manche nicht mit dem Standard kompatible Browser (z.B. IE 6) der Rollover Effect. Rollover Effect bedeutet, dass immer diejenige Tabellenzeile hervorgehoben wird, über die die Maus gerade fährt.

Zwei verschiedene Ansätze sind denkbar, um eine Tabelle zu implementieren, die diese Funktionalitäten bietet: Der eine ist der Griff zu einer Non-Standard-Komponente. Da aus oben genannten Gründen die Verwendung der Standard-JSF *h:dataTable* viele Vorteile hat, besteht der andere Ansatz darin, der Standard-JSF *h:dataTable* die benötigte Funktionalität mithilfe der Script Injection hinzuzufügen.

Script Injection heißt also, dass ein JavaScript über die gerenderte HTML-Seite läuft, sich bestimmte Elemente heraussucht und diese dann auf HTML-Ebene nachträglich verändert. An das Ende der JSF-Seite wird dazu ein JavaScript gesetzt, das beim Laden der Seite startet und die benötigten Event Listener an die entsprechenden Elemente hängt. Das hierzu nötige JavaScript findet die Elemente, indem es in der ID der Komponenten einen String sucht, der einer Naming Convention entspricht. Für ein bestimmtes gewünschtes Verhalten wird dazu ein bestimmter Namens-String festgelegt. Die JSF ID der Komponente, zu der das Verhalten hinzugefügt werden soll, muss den festgelegten Namens-String enthalten. Das Script am Ende der Seite iteriert dann über alle Elemente der HTML-Seite, für die die Naming Convention gültig ist (z.B. über alle TABLE-Elemente), und injiziert das Verhalten in alle Elemente, deren ID den festgelegten String als Teil-String enthält.

Unter [2] oder von der Quellcode-CD kann ein Script heruntergeladen werden, das vier verschiedene Naming Conventions zur Verfügung stellt: Rollover Effect, Single Row Selection, Multiple Row Selection und RadioGroups. Alle sind leicht in JSF-Seiten einzubauen und verbessern die Usability.

- *Rollover Effect*: Während der Mauszeiger über die Zeile bewegt wird, ändert sich die Style Class der Zeile. So ein Rollover Effect sieht klasse aus und der Benutzer hat einen guten Überblick über die ganze Tabellenzeile.
- *Single Row Selection* und *Multiple Row Selection*: Hiermit werden Tabellenzeilen auswählbar und verändern ihr Aussehen, wenn sie angeklickt sind.
- *RadioGroups*: Mit dieser Naming Convention können endlich auch mit JSF-Standardkomponenten RadioGroups in Tabellen verwendet werden.

Implementierungs-Pattern 4: Stacked Table

Baumstrukturen sind die Königsdisziplin der Webentwicklung. Die verschachtelte Datenstruktur führt dazu, dass eine große Menge an Parametern mit dynamisch erzeugten IDs benötigt wird, um auf allen Ebenen des Baums Eingabefelder zu implementieren. Mit einfachen JSPs ohne Framework führt dies zu kompliziertem und verschachteltem Code, den keiner gerne liest und pflegt.

Es gibt einige JSF-Komponenten, die entwickelt wurden, um Baumstrukturen darzustellen. Ich rate davon ab, eine solche Komponente zu verwenden, wenn die Komponente der Anwendung ein eigenes Objektmodell aufzwingt, anstatt mit einer einfachen Liste, die als Elemente wiederum Listen hat, zu arbeiten. Wenn die Komponente spezielle Basisklassen fordert, entsteht eine Abhängigkeit des Anwendungsmodells von der Komponente.

Wenn hingegen ineinander verschachtelte Listen als Modell verwendet werden, gibt es eine einfache und hochflexible Art, den Baum mit Standardkomponenten zu rendern: Die *h:dataTable* ist schachtelbar! Eine Spalte einer *h:dataTable* kann also wieder eine *h:dataTable* enthalten, die die Kindknoten rendert. Dieser Weg, einen Baum darzustellen, ist nicht nur einfach zu implementieren, sondern erhält gleichzeitig die volle Flexibilität des JSF-Standards.

Auf- und zuklappbare Bäume mit Stacked Table Pattern

Um das Auf- und Zuklappen der Baumknoten zu realisieren, wird jedes *panelGrid*, das einen Baumknoten enthält, von einem *h:commandLink* umschlossen, dessen Action eine Methode des Baumknotens triggert, die den Klappzustand des Knotens ändert. Das *rendered*-Attribut der Tabelle wird anhand des Klappzustands des Baumknotens gesteuert. Ein Klick auf den Baumknoten führt also dazu, dass das

Listing 2: CSS Styles, um Abstände zwischen Tabellen zu entfernen

```
table {
 table-layout: fixed;
 border-collapse: collapse;
 border: 0px;
 width:100%;
}

tr {
 border: 0px;
}

td {
 border: 0px;
 padding:0px;
 height:20px;
}
```

rendered-Attribut der Tabelle getoggelt (also von *true* auf *false* und zurück gesetzt) wird. Um einen Knoten auf- und zuzuklappen, wird ein Roundtrip durchgeführt. Diese Art, einen Baum darzustellen, toggelt die Baumknoten auf Serverseite.

Styling von Stacked Tables

Grundsätzlich sind hier die gleichen Styles zu verwenden wie beim Styling von External Header (Listing 2). Es gibt aber einen Fallstrick, der die korrekte Darstellung im IE6 verhindert: Die Kindknoten befinden sich jeweils in einer gemeinsamen Zelle mit ihrem Elternknoten. Da die Höhe der Zellen generell auf 20 px begrenzt ist, werden sie nie angezeigt: Die Zelle kann wegen des Styles *table-layout:fixed* ihre Höhe nicht dynamisch verändern. Der Ausweg besteht darin, die Zellhöhen in der *h:dataTable* auf 100 % zu setzen. Jede *h:dataTable* erhält also zusätzlich ein Attribut *columnClasses="nodeCol"* mit dem CSS Style:

```
td.nodeCol {
 height:100%;
}
```

Die CSS Styles der Spalten sorgen dafür, dass die tieferen Baumebenen eingerückt erscheinen.

Bäume mit Grafiken ansprechender darstellen

Noch schöner sieht das Ganze aus, wenn ein kleines Bild mit einem Plus(+)-Zeichen und eines mit einem Minus(-)-Zeichen hinzugefügt wird. In der ersten Spalte jedes Knotens wird

```
<h:outputText value=" " />
```

durch

Listing 3: Baum mit Standard-JSF-Komponenten

```
public class Node {
 private boolean open;
 private List children;
 public boolean isOpen() {
  return open;
 }
 public List getChildren() {
  return children;
 }
 public void setChildren(List children) {
  this.children = children;
 }
 public int getChildCount() {
  return children = null ? null : children.size();
 }
 public String toggle() {
  open = !open;
 }
}
```

```
<h:panelGrid columns="3" columnClasses="col1, col2, col3">
 <h:outputText value="Header1" />
 <h:outputText value="Header2" />
 <h:outputText value="Header3" />
</h:panelGrid>
<h:commandLink action="#{tree.toggle}">
 <h:panelGrid columns="3" columnClasses="col1, col2, col3">
  <h:outputText value="#{tree.property1}" />
  <h:outputText value="#{tree.property2}" />
  <h:outputText value="#{tree.property3}" />
 </h:panelGrid>
</h:commandLink>
<h:dataTable value="#{tree.children}" var="itemLevel1"
      rendered="#{tree.open}" columnClasses="nodeCol">
 <h:column>
  <h:commandLink action="#{itemLevel1.toggle}">
   <h:panelGrid columns="4"
columnClasses="indentLevel1, col1Level1, col2, col3">
    <h:outputText value=" " />
    <h:outputText value="#{itemLevel1.property1}">
    <h:outputText value="#{itemLevel1.property2}">
    <h:outputText value="#{itemLevel1.property3}">
   </h:panelGrid>
  </h:commandLink>
  <h:dataTable value="#{ itemLevel1.nodes}" var=
          "itemLevel2" rendered="#{itemLevel1.open &&
    itemLevel1.childCount>0}" columnClasses="nodeCol">
   <h:column>
    <h:commandLink action="#{itemLevel2.toggle}">
     <h:panelGrid columns="3"
columnClasses="indentLevel2, col1Level2, col2, col3">
      <h:outputText value=" " />
      <h:outputText value="#{itemLevel2.property1}" />
      <h:outputText value="#{itemLevel2.property2}" />
      <h:outputText value="#{itemLevel2.property3}" />
     </h:panelGrid>
    </h:commandLink>
    <h:dataTable value="#{itemLevel2.children}" var=
          "itemLevel3" rendered= "#{itemLevel2.open &&
    itemLevel2.childCount>0}" columnClasses="nodeCol">
     <h:column>
      <h:panelGrid columns="3" columnClasses=
                "indentLevel3, col1Level3, col2, col3">
       <h:outputText value=" " />
       <h:outputText value="#{itemLevel3.property1}" />
       <h:outputText value="#{itemLevel3.property2}" />
       <h:outputText value="#{itemLevel3.property3}" />
      </h:panelGrid>
     </h:column>
    </h:dataTable>
   </h:column>
  </h:dataTable>
 </h:column>
</h:dataTable>
```

Listing 4: CSS Styles zur Einrückung von Baumebenen

```
td.nodeCol {
 height:100%;
}

td.col1{
 width:100px;
}

td.indentLevel1{
 width:10px;
}

td.col1Level1{
 width:90px;
}

td.indentLevel2{
 width:20px;
}

td.col1Level2{
 width:80px;
}

td.indentLevel3{
 width:30px;
}

td.col1Level3{
 width:70px;
}
```

```
<h:graphicImage url="/images/plus.jpg" rendered=
    "#{!(item.open && item.childCount>0)}"
<h:graphicImage url="/images/minus.jpg" rendered=
    "#{item.open && item.childCount>0}"
```

ersetzt. Mit ein paar zusätzlichen Liniengrafiken können auch Linien von jedem Elternknoten zu seinen Kindern dargestellt werden.

Bäume mit clientseitigem Klappeffekt

Um das Toggeln nicht wie hier beschrieben auf Serverseite, sondern auf Clientseite (also per JavaScript im Browser) darstellen zu können, wird zusätzliches JavaScript benötigt,das beim Klick auf einen Knoten einen DIV Layer sichtbar oder unsichtbar macht, der die dazugehörige Tabelle enthält. Im Internet sind verschiedene solcher Panel- und CollapsiblePanel- Komponenten erhältlich. Als Header des Panels wird jeweils der Baumknoten angezeigt und als Klappbereich (Body) wird die innere Tabelle verwendet. Unter [3] sind Beispiele zum Toggeln zu finden.

Implementierungs-Pattern 5: Script Binding

Um JSF Tags zusammen mit JavaScript zu verwenden, benötigt man die IDs der angesprochenen HTML-Elemente. Wenn JSF HTML-Code erzeugt, vergibt es jedoch die IDs automatisch. Die Problematik, die durch das Script Binding Pattern gelöst wird, besteht also darin, an die IDs, die JSF generiert, heranzukommen, um sie im JavaScript-Code zu verwenden.

Im einfachsten Fall kann man sein Wissen über die Generierung der IDs verwenden und die IDs so, wie sie im HTML-Code vorkommen werden, in das JavaScript aufnehmen. Dieses Vorgehen hat aber einige Nachteile:

- Veränderungen der Seitenstruktur ziehen Veränderungen der IDs nach sich
- Bei Code, der per *<%@ include %>* als Snippet oder auch als Facelet in die Seite gelangt, hängt die generierte ID vom umgebenden Code ab
- Die Art der ID-Generierung ist von außen konfigurierbar (das Trennzeichen zwischen den ID-Teilen)
- Innerhalb von Tabellen werden pro Zeile eigene IDs generiert, sodass die Lösung nicht innerhalb von Tabellenzeilen funktioniert

Die Tomahawk-Bibliothek bietet eigene Varianten der JSF-Standardkomponenten an, die ein Attribut *forceId="true"* haben. Damit kann man erzwingen, dass JSF keine ID generiert, sondern die ID, die man im Tag angegeben hat, unverändert in den HTML-Code schreibt. Doch auch diese Lösung ist nicht der Weisheit letzter Schluss. Schließlich verwendet man damit ja wieder keine Standardkomponenten und in Tabellen funktioniert die Lösung auch nicht.

Die Lösung besteht darin, das Componentbinding zu verwenden, um an die clientId heranzukommen. Die UIComponent, die zu einem Tag gehört, weiß nämlich ganz genau, welche ID gerendert wird. Das Ziel-Tag, das per JavaScript manipuliert werden soll, erhält dazu das Attribut *binding="#{myBean.myBinding}"*. Die Backing Bean *myBean* erhält eine Methode *getClientIdForBinding()*, die zur UIComponent *myBinding* die clientId zurückgibt. Das JavaScript selbst wird mithilfe des *h:outputText*-Tags erzeugt und kann so innerhalb des JavaScript-Ausdrucks auf *#{myBean.clientIdForBinding}* zugreifen. Unter [3] und auf der Quellcode-CD ist die Implementierung einer Combobox mit JSF-Standardkomponenten zu finden. Dabei werden 4 Script Bindings zugleich verwendet: Ein Eingabefeld macht beim Event *onKeyDown* eine Auswahltabelle sichtbar. Neben dem Eingabefeld steht ein kleines Pluszeichen, das bei *onClick*

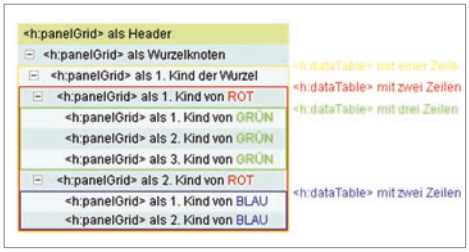

Abb. 2: Geschachtelte *h:dataTable*-Komponenten

die Auswahltabelle ebenfalls sichtbar macht. Ist die Tabelle sichtbar, wird das Plus- durch ein Minuszeichen ersetzt. Das Anklicken des Minuszeichens oder die Auswahl eines Wertes aus der Tabelle machen diese wieder unsichtbar.

Ausblick

Wer diese Patterns kennt, kann so gut wie alle Anwendungsfälle mit den Standardkomponenten von JSF implementieren. Für Spezialeffekte wie einen Date Picker können dank des Script Binding Pattern normale JavaScript-Bibliotheken verwendet werden. Dabei werden Kombinationen von Standardkomponenten immer wieder auftreten und es wäre gut, diese in Templates ablegen zu können. Die Möglichkeit, solche Templates zu definieren, bietet die Facelets-Technologie. Da Facelets Teil des JSF-2.0-Standards werden sollen, wird es dann 2 Stufen von Komponentenbibliotheken geben: Die Low Level JSF Taglibrary, die aus den im Standard definierten Komponenten besteht, und High Level Facelets Taglibraries, die die Low-Level-Komponenten auf immer neue Weise kombinieren. Ich freue mich auf Facelets-Template-Bibliotheken für alle HTML-Oberflächenkomponenten. ■

Ganesh Jung ist freiberuflicher Softwareentwickler, hat die Architekturen von mehreren JSF-Anwendungen für eine deutsche Großbank entworfen und an deren Entwicklung mitgearbeitet. In seiner Freizeit unterstützt er die Open Source Community. Bitte richten Sie Fragen zu diesem Artikel an das unter [5] angegebene Forum.

Links & Literatur

[1] www.j4fry.org/resources/Bakk_Claudia_Leeb.pdf
[2] www.j4fry.org/scriptinjection.shtml
[3] www.j4fry.org/J4Fry_Quick_Setup_Tomcat_MyFaces_Hibernate/index.faces
[4] Installationsanleitung zu [3]: www.j4fry.org/cookbook.shtml
[5] Fragen zum Artikel bitte unter https://sourceforge.net/forum/forum.php?forum_id=730375