Verwendung und Anpassung des NatTable Widgets in wenigen Schritten

NatTable 2.2

Die Erstellung großer multifunktionaler Grids kann mit SWT und JFace schnell sehr kompliziert werden, vor allem, wenn Funktionalitäten benötigt werden, die von weit verbreiteten Tabellenkalkulationsprogrammen bekannt sind. Dann stößt man sehr schnell an seine Grenzen. Eine komfortable und umfangreiche Alternative ist das SWT-Tabellen-/Grid-Widget NatTable [1], das unter der EPL auf SourceForge angeboten wird [2]. Es bietet viele Funktionalitäten out of the Box, wie zum Beispiel das Fixieren von Zeilen und Spalten, die Gruppierung von Spalten, das Ein-/Ausblenden von Spalten, die performante Verarbeitung großer Datenmengen und vieles mehr. Das neue Release 2.2 und die geplante Aufnahme des NatTable Widgets im Eclipse-Nebula-Projekt sind ein guter Grund, es genauer zu betrachten.

von Dirk Häußler

Die NatTable ist ein Tabellen-/Grid-Widget, das durch seine enorme Anpassungsfähigkeit, den großen Funktionsumfang und vor allem wegen der Fähigkeit, mit großen Datenmengen performant umgehen zu können, heraussticht. Dabei ist es mehr als nur eine einfache Tabelle, sondern eher ein Framework, um leistungsstarke Grids an die eigenen Bedürfnisse angepasst zu erstellen. Hierfür wurde ein Layer-Konzept verwendet, in dem jede Funktionalität über einen eigenen Layer realisiert wird. Bei der Erstellung eines Grids kommt entsprechend diesem Konzept ein *GridLayer* zum Einsatz. Der GridLayer wiederum ist in die vier Regionen *Body*, *ColumnHeader*, *RowHeader* und *Corner* aufgeteilt. Jede Region wird über einen *LayerStack* aufgebaut, der 1 bis n Layer enthalten kann, um die gewünschten Funktionen zu verwenden (**Abb. 1**).

Die unterste Ebene jedes LayerStacks bildet ein *DataLayer*, der die Zugriffsschicht auf die darzustellenden Daten bildet. Auf den DataLayer werden weitere Layer gesetzt, um die gewünschten Funktionalitäten einzubinden. Grundsätzlich zeichnet sich die NatTable selbst, ohne auf Standard-Widgets zurückzugreifen. Nur an wenigen Stellen kommen diese zum Einsatz, wie zum Beispiel das Text-Widget im *Bearbeiten*-Modus. Außerdem ist die NatTable eine so genannte „Virtual Table". Dies bedeutet, dass nur die Daten in der Oberfläche gerendert werden, die sichtbar sind. Aktualisierungen der Ansicht sind daher sehr schnell, und das Handling großer Datenmengen wird nicht durch UI-Prozesse verlangsamt. Dies wird entsprechend des erwähnten Layer-Konzepts über einen *ViewportLayer* auf oberster Ebene des BodyLayer-Stacks gewährleistet.

Einfaches Beispiel

Ein einfacher Grid, der Daten darstellt, ist mithilfe der NatTable mit sehr wenig Aufwand in eine RCP-Anwendung einzubinden (Listing 1). Dabei muss man sich allerdings mit den Standardeinstellungen für die Darstellung zufrieden geben, die nicht dem bekannten Look and Feel einer RCP-Anwendung entsprechen (**Abb. 2**).

Hierfür muss lediglich eine Instanz der NatTable mit einem *DefaultGridLayer* erstellt werden. Dieser benötigt wiederum als Parameter im Konstruktor die Liste der darzustellenden Daten und Informationen für den Zugriff per Reflection auf diese Daten. Mit einer Zeile Code plus der Erstellung der Informationsobjekte wird ein Grid erstellt, der Daten darstellt und verschiedene Funktionalitäten bereitstellt, z. B. das Selektieren von Zellen, Zeilen und Spalten, das Verschieben von Spalten und die Möglichkeit, Spaltenbreiten und Zeilenhöhen zu verändern.

Datenzugriff

Der Datenzugriff in der NatTable wird innerhalb eines *DataLayer* über einen *IDataProvider* realisiert. Im Normalfall verwendet man den mitgelieferten *ListDataProvider*, der mit der Liste der darzustellenden Daten und einem *IColumnPropertyAccessor* instanziiert wird. Für eine Liste von POJOs kann hier der Einfachheit halber der bereits vorhandene *ReflectiveColumnPropertyAccessor*

verwendet werden. Wie der Name vermuten lässt, greift dieser per Reflection auf die Daten zu. Hierfür benötigt er ein String-Array mit den Namen der Eigenschaften, die in der NatTable dargestellt werden sollen. Der *DefaultGridLayer* aus Listing 1 bedient sich genau dieser Mechanik, wie in Listing 2 dargestellt.

Um den Zugriff auf die anzuzeigenden Daten anzupassen, kann ein eigener *IColumnPropertyAccessor* implementiert und an den *ListDataProvider* übergeben werden. Dies ist notwendig, wenn statt einfacher POJOs komplexe Objekte dargestellt werden sollen. Bei der Implementierung des Interface müssen sowohl die vorgegebenen Methoden für den lesenden und schreibenden Zugriff auf die Properties als auch diverse Hilfsmethoden erstellt werden. Listing 3 stellt eine Implementierung des *IColumnPropertyAccessor*-Interfaces dar.

Für statische Objekte ist es sinnvoll, die Spaltenpositionen und das Mapping von Spaltenindex zu Property wie in Listing 3 als Konstanten zu halten. Für dynamische Inhalte wie beispielsweise eine Liste von Eigenschaften innerhalb der anzuzeigenden Objekte kann der Zugriff selbstverständlich auch über einen Algorithmus umgesetzt werden.

Möchte man einen eigenen *IColumnPropertyAccessor* verwenden, kann der *DefaultGridLayer* nicht genutzt werden, da er den *ReflectiveColumnPropertyAccessor* verwendet. Stattdessen muss eine eigene Unterklasse von *GridLayer* erstellt werden, damit die Anpassungen vorgenommen werden können. Dabei muss darauf geachtet werden, dass der *IDataProvider* für die Darstellung im Body und der *IDataProvider* des *ColumnHeader* auf das gleiche Datenmodell referenzieren. Aufgrund der unterschiedlichen Regionen und der darzustellenden Daten sind die beiden *IDataProvider* zwar technisch voneinander getrennt, bezüglich des Zugriffs auf die Daten aber logisch miteinander verknüpft. Alternativ kann auch ein eigener *IDataProvider* implementiert werden, z. B. wenn die Daten in einem mehrdimensionalen Array vorliegen.

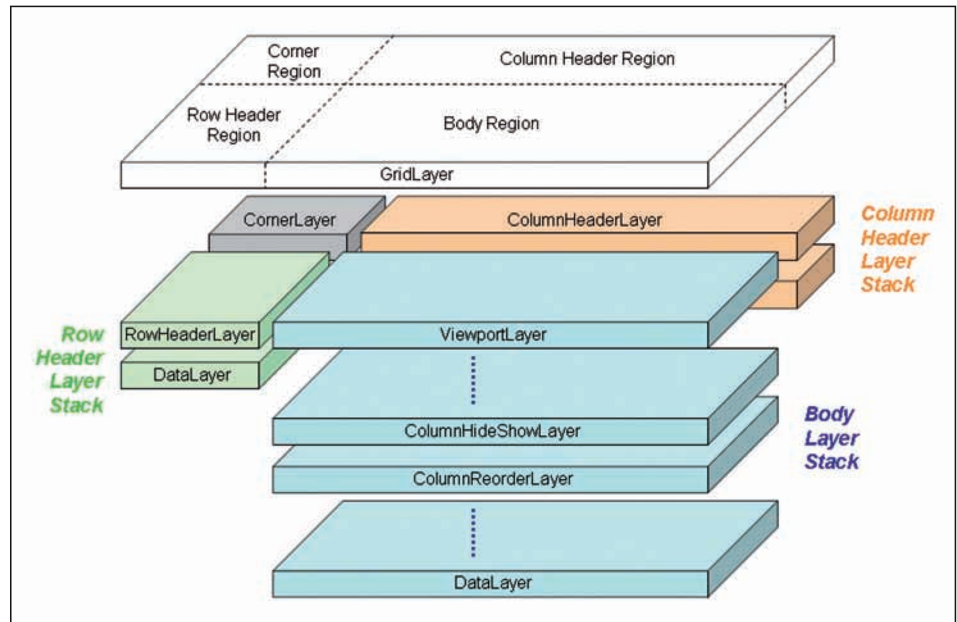

Abb. 1: Aufbau eines Grids mit der NatTable

	Firstname	Lastname	Gender	Married	Birthday
1	Lenny	Simpson	MALE	false	Thu Aug 15 00:00...
2	Lenny	Lovejoy	MALE	true	Mon Jul 26 00:00:...
3	Jessica	Simpson	FEMALE	false	Thu Mar 01 00:00...
4	Marge	Lovejoy	FEMALE	false	Thu Apr 16 00:00...
5	Edna	Lovejoy	FEMALE	true	Fri Feb 09 00:00:...
6	Carl	Leonard	MALE	false	Tue Jan 06 00:00...
7	Homer	Leonard	MALE	true	Wed Jul 13 00:00...
8	Edna	Krabappel	FEMALE	true	Sun Aug 03 0...
9	Lenny	Lovejoy	MALE	true	Mon Apr 11 00:00...
10	Jessica	Lovejoy	FEMALE	true	Mon Mar 06 00:0...

Abb. 2: Verwendung der NatTable mit Standardeinstellungen

Konfiguration

Die NatTable kann in sehr vielen Bereichen den eigenen Bedürfnissen angepasst werden. Hierzu zählen Anpassungen der Darstellung, Einstellungen für den Editiermodus und Key-/Mouse-Bindings. Um diese Konfigurationen vornehmen zu können, enthält jede NatTable-Instanz die Konfigurationscontainer *ConfigRegistry* und *UIBindingRegistry*. Für Anpassungen der Darstellung und Einstellungen des Editiermodus können Konfigurationen an der *ConfigRegistry* eingetragen werden. Eigene Key-/Mouse-Bindings werden in der *UIBindingRegistry* eingetragen. Konfigurationen werden anhand des *IConfiguration*-Interface erstellt. Als abstrakte Implementierungen stehen die *AbstractRegistry*-

Notwendige Plug-ins

Um die NatTable mit der GlazedList verwenden zu können, benötigen Sie folgende Libraries:

- glazedlists_java15-1.8.0.jar
- net.sourceforge.nattable.core-2.2.0.jar
- net.sourceforge.nattable.extension.glazedlists-2.2.0.jar

Legen Sie die JAR-Dateien in das Eclipse-*dropins*-Verzeichnis, starten Sie Eclipse neu und fügen Sie diese Plug-ins als Abhängigkeiten zu Ihrem Projekt in der *plugin.xml* hinzu. Um sich tiefer gehend mit der NatTable zu beschäftigen und die Standardimplementierungen verschiedener Layer und ihrer Konfigurationen im Detail betrachten zu können, sollten zusätzlich die Sources für die NatTable und die NatTable-Extension in das *dropins*-Verzeichnis gelegt werden. Die genannten JARs finden Sie online unter http://publicobject.com/glazedlists/, http://sourceforge.net/projects/nattable/files/NatTable/ oder auf der Heft-CD.

Configuration und die *AbstractUiBindingConfiguration* zur Verfügung. Innerhalb einer solchen Konfiguration werden über die dort zu implementierenden Methoden die einzelnen Einstellungen am entsprechenden Konfigurationscontainer registriert. Zudem gibt es über die *AggregateConfiguration* die Möglichkeit, verschiedene Konfigurationen zusammenzufassen. So erstellte Konfigurationen können sowohl der NatTable-Instanz selbst als auch den darunterliegenden Layern hinzugefügt werden (**Abb. 3**).

Um eine Anpassung in der *ConfigRegistry* einzutragen, wird über *registerConfigAttribute()* dem *ConfigAttribute* der gewünschte Wert zugewiesen. Über zwei weitere Parameter kann der *ConfigRegistry* mitgeteilt werden, dass dieses *ConfigAttribute* nur in speziellen Fällen aktiviert werden soll. Zum einen kann eingestellt werden, für welchen *DisplayMode* das *ConfigAttribute* aktiv sein soll. Hierfür sind die Werte *NORMAL*, *SELECT* oder *EDIT* für den entsprechenden Anzeigemodus möglich. Zum anderen kann das *ConfigAttribute* an selbst definierte Labels gebunden werden. Um spezielle Aktionen auf bestimmte Benutzerinteraktionen auszuführen, können an die *UiBindingRegistry* über diverse Methoden Events an Actions gebunden werden.

Die meisten Layer verfügen über eine Standardkonfiguration vom Typ *AggregateConfiguration*. Die Verwendung der Standardkonfiguration kann über einen Parameter im Konstruktor des Layers abgeschaltet werden. Nur wenn der Layer ohne die Verwendung der Standardkonfiguration erstellt wurde, ist es möglich, stattdessen eine eigene Konfiguration über die Methode *addConfiguration()* hinzuzufügen. Nachfolgend werden einige dieser Konfigurationsmöglichkeiten anhand von Beispielen genauer betrachtet.

Anpassung der Darstellung

Die Darstellungen des *ColumnHeader* und des *RowHeader* entsprechen standardmäßig nicht dem gewohnten Look and Feel einer RCP-Anwendung. Um diesen Umstand zu ändern, müssen wie bereits erwähnt entsprechende Style-Konfigurationen erstellt und den Layern übergeben werden. Die einfachste Variante hiefür ist es, die existierenden Standardkonfigurationen zu erweitern und innerhalb eines Initialisierungsblocks die vordefinierten Darstellungseigenschaften anzupassen (Listing 4). Um diese Konfigurationen zu aktivieren, müssen sie der entsprechenden Layer-Konfiguration hinzugefügt werden (Listing 5).

Listing 1

```
...
//create simple NatTable with DefaultGridLayer as underlying layer that shows
                                                                      Persons

return new NatTable(parent,
        new DefaultGridLayer(PersonService.getPersons(10), propertyNames,
                                                       propertyToLabelMap));
...
```

Listing 2

```
...
new ListDataProvider<Person>(PersonService.getPersons(), new
            ReflectiveColumnPropertyAccessor<Person>(propertyNames));
...
```

Listing 3

```
public class PersonWithAddressColumnPropertyAccessor implements
                        IColumnPropertyAccessor<PersonWithAddress> {

  @Override
  public Object getDataValue(PersonWithAddress rowObject, int columnIndex) {
    switch (columnIndex) {
      case DataModelConstants.FIRSTNAME_COLUMN_POSITION:
        return rowObject.getFirstName();
      ...
    }
    return "";
  }

  @Override
  public void setDataValue(PersonWithAddress rowObject, int columnIndex, Object
                                                                newValue) {
    switch (columnIndex) {
      case DataModelConstants.FIRSTNAME_COLUMN_POSITION:
        rowObject.setFirstName((String)newValue);
        break;
      ...
    }
  }

  @Override
  public int getColumnCount() {
    return DataModelConstants.PERSONWITHADDRESS_NUMBER_OF_COLUMNS;
  }

  @Override
  public String getColumnProperty(int columnIndex) {
    return DataModelConstants.PERSONWITHADDRESS_PROPERTY_
NAMES[columnIndex];
  }

  @Override
  public int getColumnIndex(String propertyName) {
    return Arrays.asList(DataModelConstants.
            PERSONWITHADDRESS_PROPERTY_NAMES).indexOf(propertyName);
  }
}
```

Möchte man eine eigene Konfiguration von Grund auf neu aufbauen, sollte man sich mit den möglichen Attributen und Werten vertraut machen. Leider existiert hierfür noch keine Dokumentation, weshalb an dieser Stelle nur die Möglichkeit besteht, im Quellcode der NatTable-Beispiele oder im NatTable-Quellcode selbst nachzuschlagen.

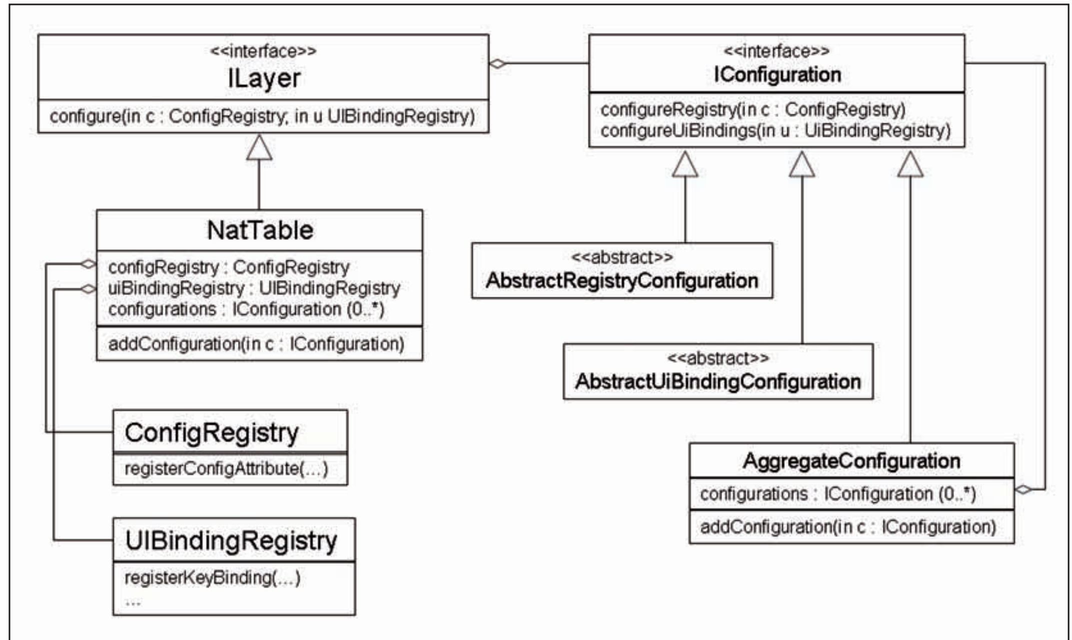

Abb. 3: Vereinfachtes Modell der NatTable-Konfigurationsarchitektur

Anpassung der Selektionsdarstellung

Die Anpassung der Darstellung der Selektion erfolgt ebenfalls anhand der oben genannten Schritte. Da der *SelectionLayer* allerdings innerhalb des *BodyLayerStacks* aufgebaut wird, kann der *DefaultBodyLayerStack* nicht mehr verwendet werden. Aus diesem Grund muss ein eigener *BodyLayerStack* erstellt werden. Der Einfachheit halber sollte man diesen nach dem Beispiel des *DefaultBodyLayerStacks* aufbauen. Wie bereits beschrieben, muss bei der Erstellung des *SelectionLayer* darauf geachtet werden, dass nicht die Standardkonfiguration verwendet wird, damit die eigene Konfiguration hinzugefügt werden kann. Auch hier empfiehlt es sich, die vorhandene Standardkonfiguration zu erweitern und nur die Darstellungskonfiguration zu überschreiben. Somit ist sichergestellt, dass der bereits vorhandene Funktionsumfang bestehen bleibt.

Im *StyledNatTableExample* auf der Heft-CD wird neben dem Überschreiben der Style-Eigenschaften auch eine zusätzliche Einstellung vorgenommen. Die Entwickler der NatTable sind in ihrer Standardimplementierung davon ausgegangen, dass Row- und Column-Header bei einer Selektion immer gleich dargestellt werden. In diesem Beispiel soll allerdings die Selektion im Row-Header kursiv dargestellt werden. Dieses Verhalten wird erreicht, indem zusätzlich die Methode *configureHeaderHasSelectionStyle()* überschrieben wird, um unterschiedliche Styles für die beiden Header zu verwenden.

NatTable Dependencies

Sollten Sie ein Projekt unter Verwendung der NatTable nicht in einer OSGi-Umgebung erstellen wollen, achten Sie darauf, dass folgende Dependencies in den Einstellungen eingetragen werden:

- org.eclipse.jface
- org.eclipse.swt
- org.apache.commons.lang
- org.apache.commons.logging
- org.eclipse.core.runtime
- org.eclipse.core.commands

Möchten Sie die GlazedLists verwenden, benötigen Sie ebenfalls *ca.odell.glazedlists*.

Listing 4

```
public class ColumnHeaderStyleConfiguration extends
DefaultColumnHeaderStyleConfiguration {
  {
    this.font = GUIHelper.DEFAULT_FONT;
    this.bgColor = GUIHelper.COLOR_WIDGET_BACKGROUND;
    this.fgColor = GUIHelper.COLOR_WIDGET_FOREGROUND;
    this.hAlign = HorizontalAlignmentEnum.CENTER;
    this.vAlign = VerticalAlignmentEnum.MIDDLE;
    this.borderStyle = null;

    this.cellPainter = new BeveledBorderDecorator(new TextPainter());
  }
}
```

Listing 5

```
ColumnHeaderLayer columnHeaderLayer =
  new ColumnHeaderLayer(columnHeaderDataLayer, bodyLayer, selectionLayer, false);
columnHeaderLayer.addConfiguration(new DefaultColumnHeaderLayerConfiguration() {
  @Override
  protected void addColumnHeaderStyleConfig() {
    addConfiguration(new ColumnHeaderStyleConfiguration());
  }
});
```

	Firstname	Lastname	Gender	Married	Birthday	Street	Housenumber	Postal Code	City
1	Waylon	Smithers	MALE	☑	Tue Sep 09 00:00...	Plympton Street	168	44444	Waverly Hills
2	Homer	Simpson	MALE	☐	Thu Jun 03 00:00...	South Street	147	11111	Springfield
3	Homer	Leonard	MALE	☐	Tue Jan 09 00:00...	Highland Avenue	86	11111	Springfield
4	Lenny	Carlson	MALE	☐	Wed Aug 11 00:0...	Main Street	168	33333	Ogdenville
5	Helen	Simpson	FEMALE	☐	Wed Sep 11 00:0...	South Street	146	22222	Shelbyville
6	Homer	Krabappel	MALE	☐	Sun Jul 23 00:00:...	Main Street	54	33333	Ogdenville
7	Edna	Lovejoy	FEMALE	☑	Thu Nov 15 00:00...	Main Street	128	55555	North Haverbrook
8	Waylon	Leonard	MALE	☐	Thu Apr 10 00:00...	Plympton Street	181	55555	North Haverbrook
9	Helen	Lovejoy	FEMALE	☑	Tue Nov 28 00:00:...	South Street	35	55555	North Haverbrook
10	Marge	Lovejoy	FEMALE	☑	Sat Aug 02 00:00...	Plympton Street	38	22222	Shelbyville

Abb. 4: NatTable-Ansicht mit Anpassungen an die Darstellung

Bedingte Formatierung

Neben allgemeinen Einstellungen ist es auch möglich, die Darstellung von Bedingungen abhängig zu machen. Die NatTable verwendet hierfür ein Label-Konzept, das es erlaubt, anhand eigener Kriterien Labels an Zellen zu setzen. An diese Labels können wiederum spezielle Konfigurationseinstellungen gebunden werden. Diese werden dann nur für alle so markierten Zellen verwendet. Labels werden an eine Zelle im Grid über einen *IConfigLabelAccumulator* gesetzt. Neben dem *AbstractOverrider*, der sich als Basis für eigene Implementierungen eignet, gibt es diverse vorgefertigte Klassen. Leider ist auch hier die vorhandene Dokumentation sehr spärlich, weshalb oftmals der Quellcode zu Rate gezogen werden muss.

Im bereits erwähnten *StyledNatTableExample* wird ein eigener *AbstractOverrider* erstellt, um zwei bedingte Formatierungen zu erreichen. Zum einen sollen weibliche Personen über einen gelben Hintergrund hervorgehoben werden, zum anderen soll der Status, ob eine Person verheiratet oder ledig ist, über eine Checkbox dargestellt werden. Da sowohl die Spaltenposition als auch der Inhalt einer Zeile ausgewertet werden müssen, wird der *IDataProvider* benötigt, der die anzuzeigenden Daten im Body zur Verfügung stellt. In der Methode *accumulateConfigLabels()* des *AbstractOverrider* kann dann das Objekt ausgelesen und ausgewertet werden, das in der übergebenen Zeile angezeigt wird. Anhand dieser Auswertung und der Spaltenposition können dem übergebenen *LabelStack* entsprechende Labels hinzugefügt werden, die wiederum in den Konfigurationen referenziert werden können (Listing 6).

Der erstellte *CellLabelOverrider* im genannten Beispiel wird an den *DataLayer* des Body über die Methode *setConfigLabelAccumulator()* gesetzt. Damit die bedingten Formatierungen ausgewertet werden, muss außerdem eine entsprechende Style-Konfiguration erstellt werden. In dieser Konfiguration werden für den *DisplayMode.NORMAL* und die erstellten Labels neue *ConfigAttributes* registriert. Diese Style-Konfiguration wird der NatTable selbst hinzugefügt. Dabei ist wie bei den einzelnen Layern darauf zu achten, dass die Verwendung der Standardkonfiguration abgeschaltet wird,

Listing 6

```
...
public void accumulateConfigLabels(LabelStack configLabels, int columnPosition, int
                                                                      rowPosition) {
   Person rowObject = dataProvider.getRowObject(rowPosition);

   if (rowObject.getGender().equals(Person.Gender.FEMALE)) {
      configLabels.addLabel(FEMALE_LABEL);
   }

   switch (columnPosition) {
      case (DataModelConstants.MARRIED_COLUMN_POSITION):
         configLabels.addLabel(MARRIED_LABEL);
         break;
      ...
   }
}
```

Listing 7

```
...
configRegistry.registerConfigAttribute(
   EditConfigAttributes.CELL_EDITOR, new CheckBoxCellEditor(),
   DisplayMode.EDIT, CellLabelOverrider.MARRIED_LABEL);
...
configRegistry.registerConfigAttribute(
   CellConfigAttributes.DISPLAY_CONVERTER,
   new DefaultIntegerDisplayConverter(),
   DisplayMode.NORMAL,
   CellLabelOverrider.POSTALCODE_LABEL);
...
configRegistry.registerConfigAttribute(
   EditConfigAttributes.DATA_VALIDATOR,
   getPostalCodeValidator(),
   DisplayMode.EDIT,
   CellLabelOverrider.POSTALCODE_LABEL);
...
```

Tipp

Bei der Erstellung verschiedener Tabellen und Grids mit unterschiedlichen Anforderungen hat es sich als vorteilhaft erwiesen, die Layer eines Layer Stacks über entsprechende *getter* zugreifbar zu machen. Vor allem der *SelectionLayer* kommt immer wieder zum Einsatz, wenn Aktionen auf die bestehende Selektion ausgeführt werden sollen.

um das Hinzufügen eigener Konfigurationen zu ermöglichen. Damit die globalen Darstellungseinstellungen der NatTable weiterhin bestehen, muss auch darauf geachtet werden, dass die *DefaultNatTableStyleConfiguration* nach dem Ausschalten der Standardkonfiguration manuell der NatTable hinzugefügt wird. Nachdem die oben beschriebenen Konfigurationen zur Anpassung der Darstellung von Daten und Selektion und die bedingten Formatierungen hinzugefügt wurden, wird die NatTable wie in **Abbildung 4** dargestellt.

Kontextmenüs

Erstellt man einen eigenen *BodyLayerStack* nach dem Beispiel des *DefaultBodyLayerStacks*, werden neben dem *ViewportLayer* und dem *SelectionLayer* auch der *ColumnReorderLayer* und der *ColumnHideShowLayer* eingesetzt. Während die Funktionalität des *ColumnReorderLayer*, über Drag-and-drop-Spalten umzuordnen, bereits aktiv ist, ist es noch nicht möglich, Spalten aus- und wieder einzublenden. Hierfür haben die Entwickler der NatTable Kontextmenüs eingeführt. Entsprechend des Konfigurationskonzepts gibt es bereits eine von der *AbstractUiBindingConfiguration* abgeleitete Klasse *HeaderMenuConfiguration*. Diese definiert Popup-Menüs und bindet sie an den Rechtsklick im Header-Bereich des Grids. Sollte man, wie in unserem Fall, nicht alle möglichen Menüeinträge zur Verfügung stellen wollen, kann eine eigene von *HeaderMenuConfiguration* abgeleitete Klasse erstellt werden, die nur noch die gewünschten Menüeinträge enthält. Auf diese Weise ist es auch möglich, eigene Menüeinträge hinzuzufügen. Um diese Konfiguration zu aktivieren, muss sie, wie zuvor die Konfiguration für bedingte Formatierungen, der NatTable hinzugefügt werden. Leider ist auch die Erstellung eigener Menüeinträge noch nicht in der existierenden Benutzerdokumentation vorhanden, weshalb ich nochmals die Empfehlung ausspreche, die Source-Pakete mit in das *dropins*-Verzeichnis zu legen, um sich API und Beispiele ansehen zu können.

Sortierung

Die Entwickler der NatTable empfehlen die Verwendung der *GlazedLists*, um weiterführende Funktionalitäten wie die Sortierung umzusetzen [3]. Hierfür wurde sogar eine Erweiterung entwickelt, die die Anbindung der *GlazedLists* in der NatTable erleichtert. Die Ursprungsliste wird dabei in die entsprechenden *GlazedLists*-Implementierungen eingepackt, bevor ein *ListDataProvider* damit erstellt wird (weitere Informationen zu *GlazedLists* finden sie unter http://www.glazedlists.com/). Um der Darstellung mitzuteilen, dass Veränderungen an der Datenstruktur vorgenommen wurden, wie beispielsweise gelöschte oder hinzugefügte Datensätze, arbeitet die NatTable mit Events. In der Erweiterung für die *GlazedLists* findet sich der *GlazedListsEventLayer*, der *GlazedLists*-Events automatisch verarbeitet und in entsprechende NatTable-Events transformiert. Der *GlazedListsEventLayer* wird direkt auf den *DataLayer* des *BodyLayerStacks* gelegt, bevor weitere funktionale Layer hinzugefügt werden.

Damit die Sortierung über die Oberfläche gesteuert werden kann, wird auf die oberste Ebene des *ColumnHeaderLayerStacks* ein *SortHeaderLayer* gesetzt. Dieser reagiert auf Benutzereingaben im Spaltenkopf und führt die entsprechende Sortierung durch. Um diese Aufgabe erfüllen zu können, benötigt der *SortHeaderLayer* ein *ISortModel*, an das die Sortierung übertragen wird. Für die Verwendung von *GlazedLists* ist in der NatTable-Erweiterung das *GlazedListsSortModel* implementiert worden. Das Model realisiert die Sortierung über Comparators, die in einer *IConfiguration* hinterlegt wurden.

Sortierung ohne GlazedLists

Die Verwendung der *GlazedLists* ist eine Empfehlung der NatTable-Entwickler. Selbstverständlich kann diese Funktionalität auch mit anderen Listenimplementierungen erreicht werden. Hierfür muss eine eigene Implementierung des *ISortModels* erstellt und dem *SortHeaderLayer* übergeben werden. Ein Beispiel für die Umsetzung der Sortierung ohne GlazedLists ist das *SimpleSortableNatTableExample* auf der Heft-CD.

Anzeige

Mitten in Deutschland gibt's einen Ort, der in ruhiger und schöner Atmosphäre allerbeste Lernbedingungen bietet.

Admins & Entwickler: Erleben Sie, wie schnell man sich in eine Sache einarbeiten kann, wenn man wirklich einmal ein paar Tage hineintaucht. Das geht weder im eigenen Betrieb, noch abends nach der Arbeit. Auch den neuen Kollegen arbeitet niemand 'mal eben so nebenbei ein. „LERNEN" ist eine vollwertige Tätigkeit, und seit über 2000 Jahren sind kluge Leute davon überzeugt, daß man Universitäten und Schulungseinrichtungen braucht. Siehe www.linuxhotel.de

In der *DefaultSortConfiguration* wird der *DefaultComparator* gebunden, der an *Comparable.compareTo()* weiterdelegiert. Dies kann über eine eigene Konfiguration angepasst werden. Dabei ist es sogar möglich, mithilfe von Labels an jede Spalte einen eigenen Comparator zu binden.

Mit der Standardkonfiguration wird die NatTable nach einer Spalte über einen Klick auf den Spaltenkopf bei gedrückter ALT-Taste sortiert. Dabei ist es auch möglich, eine mehrstufige Sortierung durchzuführen. Um die Sortierung auf einfache Klicks im Spaltenkopf durchzuführen, bietet die NatTable die *SingleClickSortConfiguration* an. Wird diese Konfiguration anstelle der Standardkonfiguration dem *SortHeaderLayer* übergeben, wird ein intuitiveres Verhalten für die Sortierung umgesetzt.

NatTable macht es einfach, multifunktionale Grids in RCP-Anwendungen zu integrieren.

Daten bearbeiten

Neben der reinen Anzeige unterstützt die NatTable auch das Bearbeiten der dargestellten Daten. Die Daten können dabei über Textfeld, ComboBox oder CheckBox innerhalb der Tabelle selbst bearbeitet werden. Außerdem können Konverter und Validatoren eingesetzt werden, um die Korrektheit der Daten zu gewährleisten. Beim Bearbeiten wird das Datenmodell von der NatTable automatisch über den *IColumnPropertyAccessor* aktualisiert, ohne dass weitere Schritte durchgeführt werden müssen. Standardmäßig ist das Bearbeiten der Daten innerhalb der NatTable deaktiviert. Zur Aktivierung muss eine *IEditableRule* in der *ConfigRegistry* registriert werden. Dabei können sowohl die vorgefertigten *IEditableRule.ALWAYS_EDITABLE* und *IEditableRule.NEVER_EDITABLE* als auch eigene *IEditableRules* verwendet werden. Entsprechend dem Konfigurationsmechanismus können *IEditableRules* global oder über den Label-Mechanismus nur für spezielle Labels konfiguriert werden, sodass nicht der gesamte Grid, sondern nur bestimmte Spalten oder Zellen editierbar sind. Über eine selbst implementierte *IEditableRule* kann dynamisch auf die Umgebung reagiert werden, z. B. um die Bearbeitung über einen Schalter oder anhand von Berechtigungen ein- oder auszuschalten.

Ohne zusätzliche Konfigurationen können nach der Aktivierung des *Bearbeiten*-Modus die Daten über Textfelder bearbeitet werden. Um stattdessen eine CheckBox oder eine ComboBox zu verwenden, muss der entsprechende Editor über den Label-Mechanismus registriert werden (Listing 7). Das Label muss zuvor selbstverständlich an die entsprechenden Zellen gesetzt werden.

Um andere Datentypen als Strings in der NatTable anzeigen und bearbeiten zu können, werden Konverter benötigt. Für Zahlen, boolesche Werte und Datumsformate werden bereits Standardkonverter mitgeliefert. Zudem ist es möglich, eigene Konverter anhand des *IDisplayConverte*-Interface zu erstellen. Hierbei müssen Methoden zur Verfügung gestellt werden, die die Transformation von Objekt zu Darstellung und wieder zurück durchführen können.

Für die Validierung der eingegebenen Daten kann ein Validator auf Basis des *IDataValidator*-Interface erstellt werden. In der dort zu implementierenden *validate()*-Methode wird der eingetragene Wert geprüft und *true* zurückgeliefert, sollte der Wert gültig sein. Zur Verwendung müssen Konverter und Validatoren in der *ConfigRegistry* bei den gewünschten Labels registriert werden.

Aktuell werden fehlerhafte Eingaben in der NatTable schon während der Eingabe in roter Schrift dargestellt. Bei Bestätigung der fehlerhaften Eingabe wird die Eingabe verworfen und der zuvor gültige Wert wiederhergestellt. Zudem wird die Konvertierung nur durchgeführt, wenn eine zugehörige Validierung registriert wurde. An diesem Missstand wird aktuell gearbeitet und bald in einer Folgeversion behoben. Die Konfiguration für den *Bearbeiten*-Modus muss wie weiter oben beschrieben der NatTable hinzugefügt werden, um sie zu aktivieren.

Fazit

Mithilfe der NatTable ist es sehr einfach, multifunktionale Grids in eine RCP-Anwendung zu integrieren. Vor allem die vielfältigen Anpassungsmöglichkeiten machen die NatTable sehr interessant. Neben den hier vorgestellten Anpassungen und Funktionalitäten bietet die NatTable noch viele weitere Möglichkeiten, einen den eigenen Bedürfnissen angepassten Grid zu erstellen. So können beispielsweise Spalten- und Zeilengruppierungen, Filterung und Baumstrukturen eingebunden sowie weitere Layer erstellt und hinzugefügt werden. Einen guten Startpunkt, um sich weiter mit der NatTable zu befassen, bietet die Website des Projekts [1]. Dort gibt es eine Benutzerdokumentation und mehrere Beispiele, die den Einstieg zur Verwendung der NatTable erleichtern.

Dirk Häußler ist Senior Consultant bei der BeOne Stuttgart GmbH und seit mehreren Jahren im Bereich der Java-Entwicklung tätig. Er war in Projekten im Umfeld von JSF, Spring und Eclipse RCP tätig und ist seit Kurzem aktiver Committer im NatTable-Projekt.

Links & Literatur

[1] http://www.nattable.org

[2] http://sourceforge.net/projects/nattable/

[3] http://www.glazedlists.com/