Quellcode auf CD!

NatTable Advanced: Komfortfunktionen leicht gemacht

Excel kann das doch auch!

Bindet man Tabellen oder Grids in die eigenen Anwendungen ein, hört man oft von Anwendern „Warum kann ich hier nicht …?“ gefolgt von „In Excel geht das aber!“. Den Entwicklern des NatTable Widgets ist das bewusst, und so gibt es diverse Komfortfunktionen out of the Box, die bei Bedarf ohne großen Aufwand dem eigenen Grid hinzugefügt werden können. Der folgende Artikel beschäftigt sich mit einigen dieser Komfortfunktionen.

von Dirk Fauth

Das NatTable Widget [1] ist ein umfangreiches Framework, um multifunktionale Grids in die eigenen RCP-Anwendungen einzubinden. Nicht nur das Layer-Konzept, durch das jede Funktion über einen eigenen Layer separat dem Grid hinzugefügt werden kann (**Abb. 1**), sondern auch die vielfältigen Konfigurationsmöglichkeiten je Layer machen NatTable zum derzeit stärksten Tabellen-/Grid Widget in der RCP-Welt. Das ist auch der Eclipse-Community nicht entgangen, und so wurde bereits Mitte des letzten Jahres das Project Proposal für die Aufnahme in das Nebula-Projekt eingereicht. Allerdings hängt der Aufnahmeprozess aufgrund diverser anderer Themen in der Eclipse-Community noch immer in verschiedenen Review-Schritten fest. Die Weiterentwicklung von NatTable ist allerdings nicht stehen geblieben, und so ist die aktuelle Version 2.3.1.1 durch verschiedenste Bugfixes und weitere Funktionen reifer und stabiler geworden. Einen Einstieg in die Verwendung des NatTable Widgets wurde in der Ausgabe 4.2011 des Eclipse Magazins gegeben. Dieser Artikel setzt darauf auf und wird einige der Komfortfunktionen wie Filterung, Gruppierung und Bäume in NatTable beschreiben.

Mehr zum Thema

Dieser Artikel schließt an den im Eclipse Magazin 4.2011 erschienenen Artikel „NatTable 2.2“ von Dirk Häußler an.

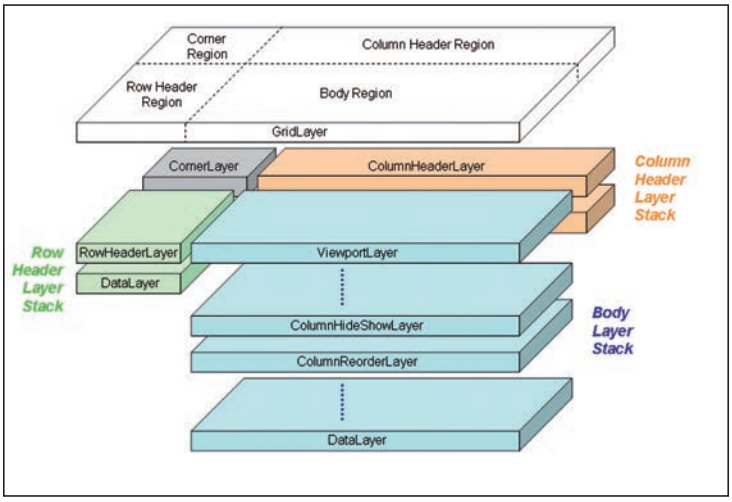

Abb. 1: NatTable-Architektur

Filter

In Tabellen und Grids, die große Datenmengen beinhalten, ist es unter Umständen sehr schwer, die gewünschten Daten zu finden. In solchen Fällen möchte man seinen Benutzern die Möglichkeit bieten, die dargestellten Daten zu filtern. Hierfür bietet NatTable eine Filterzeile an, die im *ColumnHeader* eingebunden werden kann. Da die Filterung über die Filterzeile unter Verwendung der *GlazedLists* implementiert wurde, befinden sich die notwendigen Komponenten in der *GlazedLists* Extension von NatTable.

Um die Filterzeile in den eigenen Grid einzubinden, müssen der Body und der *ColumnHeader* des Grid modifiziert werden. Im Body muss zuerst die Datenliste in einer *FilterList* eingepackt werden, bevor sie an den *IDataProvider* übergeben wird. Auf den darauf aufbauenden *DataLayer* wird anschließend der *GlazedListsEventLayer* gelegt, bevor weitere funktionale Layer eingefügt werden. Dieser verarbeitet die Events, die von der *FilterList* gefeuert werden, ohne dass sich der Entwickler selbst

noch darum kümmern müsste. Im *ColumnHeader* wird entsprechend des Layer-Konzepts NatTable das *FilterRowHeaderComposite* als weiterer Layer hinzugefügt (Listing 1). Diese beiden Anpassungen an eine bestehende NatTable-Verwendung aktivieren die voll funktionsfähige Filterzeile, wie es in **Abbildung 2** dargestellt ist.

Über den bekannten Konfigurationsmechanismus können Editoren, der Filtermodus und Converter für jede einzelne Spalte der Filterzeile konfiguriert werden. Hierfür werden jeder Zelle in der Filterzeile automatische Labels hinzugefügt, über die der Konfigurationsmechanismus die Zuordnung der Einstellungen vornehmen kann. Die Labels werden aus der Konstanten *FilterRowDataLayer.FILTER_ROW_COLUMN_LABEL_PREFIX* und der Spaltenposition zusammengesetzt. In Listing 2 ist ein Beispiel für die verschiedenen Konfigurationsmöglichkeiten der Filterzeile aufgeführt.

Wie im Body können auch in der Filterzeile anstelle des *TextCellEditor* andere Editoren wie der *ComboBoxCellEditor* verwendet werden. Das ist dann sinnvoll, wenn nur nach vordefinierten Werten gefiltert werden soll. Welche Filterlogik angewendet werden

Neuerungen seit NatTable 2.2

2.2.1

- Umsetzung der Internationalisierung mit englischen und deutschen Übersetzungen
- *VerticalTextPainter*, um Texte vertikal in NatTable darstellen zu können
- *NatGridLayerPainter*, um NatTable außerhalb des Grids im sichtbaren Bereich zu erweitern
- Unterstützung für Checkboxen im *ColumnHeader*, um alle Checkboxen in einer Spalte automatisch zu aktivieren/deaktivieren
- Anpassung des *ExportToExcelCommandHandler*, um das automatische Öffnen von Excel konfigurieren zu können

2.3

1. Anpassungen im Editierverhalten:
   - Konsolidierung des Verhaltens eines Editors, wenn der Fokus verloren wird
   - Trennung von Konvertierung und Validierung
   - Konfigurierbarkeit der Konvertierungs-/Validierungsfehlerbehandlung über die Konfigurationsattribute *EditConfigAttributes.CONVERSION_ERROR_HANDLER* und *EditConfigAttributes.CONVERSION_ERROR_HANDLER*, für die Implementierungen vom Typ *IEditErrorHandler* registriert werden können (z. B. *DialogErrorHandling*, um im Fehlerfall einen Dialog darzustellen)
   - Diverse kleinere API-Änderungen, um Kontextinformationen verfügbar zu machen. Genauere Informationen und Migrationshilfen finden Sie unter [6]
2. GroupBy-Funktion
3. Erweiterte Visualisierung beim Drag and Drop von Spalten
4. *ExtendedReflectiveColumnPropertyAccessor*, um per Reflection auch verschachtelte Objekte per Punktnotation in NatTable anzeigen zu können

2.3.1

- Apache POI Extension
- *DefaultGlazedListsStaticFilterStrategy*, um statische Filter zusammen mit FilterRow verwenden zu können

Listing 1

```
...
CompositeMatcherEditor<PersonWithAddress> autoFilterMatcherEditor =
  new CompositeMatcherEditor<PersonWithAddress>();
filterList.setMatcherEditor(autoFilterMatcherEditor);

FilterRowHeaderComposite<PersonWithAddress> filterRowHeaderLayer =
  new FilterRowHeaderComposite<PersonWithAddress>(
    new DefaultGlazedListsFilterStrategy<PersonWithAddress>(
autoFilterMatcherEditor,
columnPropertyAccessor,
configRegistry),
    columnHeaderLayer, columnHeaderDataProvider, configRegistry
  );
...
```

Listing 2

```
ICellEditor comboBoxCellEditor =
new ComboBoxCellEditor(Arrays.asList(Gender.FEMALE, Gender.MALE));
configRegistry.registerConfigAttribute(EditConfigAttributes.CELL_EDITOR,
  comboBoxCellEditor,
  DisplayMode.NORMAL,
FilterRowDataLayer.FILTER_ROW_COLUMN_LABEL_PREFIX +
                                        DataModelConstants.GENDER_COLUMN_POSITION);

configRegistry.registerConfigAttribute(TEXT_MATCHING_MODE,
  TextMatchingMode.EXACT,
  DisplayMode.NORMAL,
FilterRowDataLayer.FILTER_ROW_COLUMN_LABEL_PREFIX +
                                        DataModelConstants.GENDER_COLUMN_POSITION);

configRegistry.registerConfigAttribute(
FilterRowConfigAttributes.FILTER_DISPLAY_CONVERTER,
new DefaultIntegerDisplayConverter(),
DisplayMode.NORMAL
FilterRowDataLayer.FILTER_ROW_COLUMN_LABEL_PREFIX +
                                    DataModelConstants.HOUSENUMBER_COLUMN_POSITION);

configRegistry.registerConfigAttribute(TEXT_MATCHING_MODE,
TextMatchingMode.REGULAR_EXPRESSION,
DisplayMode.NORMAL,
FilterRowDataLayer.FILTER_ROW_COLUMN_LABEL_PREFIX +
                                  DataModelConstants. HOUSENUMBER_COLUMN_POSITION);

configRegistry.registerConfigAttribute(FilterRowConfigAttributes.TEXT_DELIMITER, "&");
```

	Firstname	Lastname	Gender	Married	Birthday	Street	Housenumber	Postal Code	City
			MALE	true			>50 & <100		
1	Bart	Flanders	MALE	☑	1953-05-13	Main Street	92	11111	Springfield
2	Timothy	Simpson	MALE	☑	1960-02-01	Plympton Street	76	44444	Waverly Hills
3	Timothy	Lovejoy	MALE	☑	1932-07-04	South Street	63	44444	Waverly Hills
4	Waylon	Krabappel	MALE	☑	1925-04-19	Plympton Street	88	33333	Ogdenville
5	Homer	Carlson	MALE	☑	2000-02-19	Oak Grove Street	90	33333	Ogdenville
6	Waylon	Lovejoy	MALE	☑	1939-04-18	Elm Street	66	22222	Shelbyville

Abb. 2: NatTable mit FilterRow (FilterableNatTableExample.java)

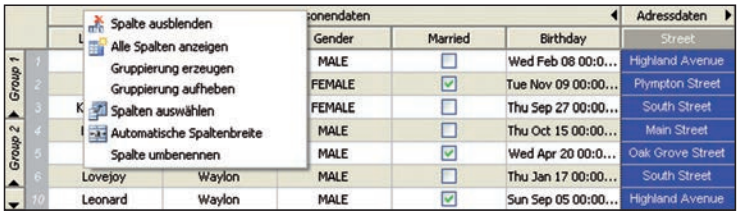

Abb. 3: NatTable mit Spalten- und Zeilengruppierungen (RowGroupingNatTableExample.java)

soll, kann über den *TEXT_MATCHING_MODE* konfiguriert werden. Entsprechend der Filtermodi, die von den *GlazedLists* angeboten werden, unterstützt die Filterzeile von NatTable die vier Filtermodi *EXACT, CONTAINS, STARTS_WITH* und *REGULAR_EXPRESSION*. Standardmäßig wird der Modus *CONTAINS* verwendet, was über die Konfiguration für jede einzelne Spalte separat verändert werden kann. Mit dem Filtermodus *REGULAR_EXPRESSION* können logische Ausdrücke beispielsweise durch die Operatoren > und < für die Filterung erstellt werden. Das ist vor allem für Spalten sinnvoll, in denen Zahlenwerte dargestellt werden. Dabei muss allerdings darauf geachtet werden, dass derselbe *IDisplayConverter* in der Filterzeile verwendet wird wie im Body, da die Filterlogik ansonsten Fehler beim Vergleich der Werte produziert. Um die logischen Ausdrücke entsprechend interpretieren zu können, muss allerdings der *IDisplayConverter* für die Filterzeile über das Konfigurationsattribut *FilterRowConfigAttributes.FILTER_DISPLAY_CONVERTER* registriert werden. Um komplexere Filterausdrücke zu ermöglichen, kann außerdem ein Trennzeichen konfiguriert werden, über das *UND*-Verknüpfungen definiert werden können. Hierfür muss das Konfigurationsattribut *FilterRowConfigAttributes.TEXT_DELIMITER* gesetzt werden.

Neben der dynamischen Filterung ist es in manchen Fällen auch notwendig, zusätzlich eine statische Filterung der Daten vorzunehmen, zum Beispiel aufgrund eines Rechtekonzepts. Um einen statischen Filter zusammen mit der Filterzeile verwenden zu können, wurde NatTable in der Version 2.3.1 *DefaultGlazedListsStaticFilterStrategy* hinzugefügt. Dieser können über die Methoden *addStaticFilter(Matcher)* und *addStaticFilter(MatcherEditor)* statische Filter hinzugefügt und über die Methoden *removeStaticFilter(Matcher)* und *removeStaticFilter(MatcherEditor)* wieder entfernt werden. *Matcher* und *MatcherEditor* sind Klassen der *GlazedLists*, um Filter auf Listen zu setzen. Die Filterzeile arbeitet mit dem speziellen *CompositeMatcherEditor*, mit dem mehrere Filter kumulativ gesetzt werden können. Genauere Informationen zur Filterung mit *GlazedLists* finden Sie unter [2].

Spaltengruppierungen

NatTable verfügt über verschiedene Möglichkeiten, Spalten und Zeilen zu gruppieren. Entsprechend dem grundlegenden Konzept von NatTable werden diese Funktionen über verschiedene Layer abgebildet. Um die Spaltengruppierung verwenden zu können, muss im *ColumnHeader* der *ColumnGroupHeaderLayer* eingebunden werden. Er kann über ein *ColumnGroupModel* konfiguriert werden, um Spaltengruppierungen zu erstellen (Listing 3). Das *ColumnGroupModel* bietet zudem die Möglichkeit, die erstellten Spaltengruppen anhand ihres Index zu sperren, sodass sie zur Laufzeit nicht wieder entfernt werden können. Außerdem können Spaltengruppen als *static* konfiguriert werden, was bedeutet, dass sie beim Einklappen der Spaltengruppe sichtbar bleiben.

Um die Spaltengruppierungen um weitere Funktionen wie das Gruppieren zur Laufzeit, das Ein-/Ausklappen und das Verschieben von Spaltengruppen zu erweitern, müssen zusätzlich im Body des Grid die beiden funktionalen Layer *ColumnGroupReorderLayer* und *ColumnGroupExpandCollapseLayer* eingebunden werden. Anschließend können die Spaltengruppen über Doppelklick ein- und ausgeklappt werden. Hierbei bleiben, wie bereits erwähnt, die im *ColumnGroupModel* als *static* konfigurierten Spalten sichtbar, mindestens aber die erste Spalte der Spaltengruppe. Um die Spaltengruppierungen zur Laufzeit konfigurieren und umsortieren zu können, müssen die notwendigen

Listing 3

```
ColumnGroupModel model = new ColumnGroupModel();
model.addColumnsIndexesToGroup("Personendaten", 0, 1, 2, 3, 4);
model.addColumnsIndexesToGroup("Adressdaten", 5, 6, 7, 8);

ColumnGroupHeaderLayer columnGroupHeaderLayer =
  new ColumnGroupHeaderLayer(columnHeaderLayer, selectionLayer, model);

//add column chooser to column header
DisplayColumnChooserCommandHandler columnChooserCommandHandler =
  new DisplayColumnChooserCommandHandler(
      selectionLayer,
      bodyLayer.getColumnHideShowLayer(),
      columnHeaderLayer,
      columnHeaderDataLayer,
      columnGroupHeaderLayer,
      model);
columnHeaderLayer.registerCommandHandler(columnChooserCommandHandler);
```

Funktionen für den Benutzer im Header-Menü verfügbar gemacht werden. Die entsprechenden Menüeinträge können über den *PopupMenuBuilder* hinzugefügt werden. Die vordefinierte *HeaderMenuConfiguration* von NatTable enthält die entsprechenden Menüeinträge bereits.

Zeilengruppierungen

Wie in **Abbildung 3** zu sehen ist, unterstützt NatTable neben den Spalten- auch Zeilengruppierungen. Zur Darstellung der Zeilengruppen muss der *RowGroupHeaderLayer* dem *RowHeader* hinzugefügt werden (Listing 4). Er verwendet in der Standardkonfiguration den normalen *TextPainter* zur Darstellung der Namen der Zeilengruppen. Um die Zeilengruppennamen vertikal darzustellen, kann der in Version 2.2.1 hinzugefügte *VerticalTextPainter* konfiguriert werden.

Die Zeilengruppen selbst werden ähnlich den Spaltengruppen über die Model-Klasse *RowGroupModel* konfiguriert. Sie benötigt den *DataProvider* des Body und ein *RowGroup*-Objekt je Zeilengruppe. Der *RowGroup* können über *addMemberRow()* die Objekte übergeben werden, die zu einer Gruppe zusammengefasst werden sollen (Listing 5).

Zeilengruppen können standardmäßig wie Spaltengruppen ein- und ausgeklappt werden. Im Konstruktor einer *RowGroup* kann über den letzten Parameter angegeben werden, ob die Zeilengruppe initial ein- oder ausgeklappt dargestellt werden soll. Der *RowGroup* kann über *setCollapseable ()* mitgeteilt werden, ob sie das Ein-/Ausklappen unterstützt oder nicht. Kann die Zeilengruppe ein-/ausgeklappt werden, so muss mindestens ein Zeilenobjekt als *static* hinterlegt werden, damit die Zeilengruppe beim Zusammenklappen noch dargestellt wird. Andernfalls verschwindet die Zeilengruppe und kann nicht wieder hergestellt werden. Im Gegensatz zu den Spaltengruppen wird hier nicht standardmäßig das erste gesetzte Objekt der Gruppe als *static* hinterlegt. Wichtig ist zudem, dass bei Aktivierung der Zeilengruppierung nur noch die Datenzeilen in NatTable dargestellt werden, die einer Zeilengruppe angehören.

Wie bei den Spaltengruppierungen müssen für die Funktion des Ein-/Ausklappens von Zeilengruppen zwei weitere funktionale Layer in den Body mit aufgenommen werden, der *RowHideShowLayer* und der darauf aufsetzende *RowGroupExpandCollapseLayer*, der das erstellte *RowGroupModel* im Konstruktor benötigt (Listing 5). Sind die Zeilengruppen konfiguriert und die Layer in den eigenen Grid eingebunden, können die nun dargestellten Zeilengruppen per Doppelklick ein- und ausgeklappt werden.

Trees

Ein weiteres mächtiges Feature von NatTable ist die Darstellung einer Baumstruktur innerhalb der Tabelle. Hierüber lassen sich die Inhalte der Tabelle gruppiert darstellen und anhand der Baumstruktur auf- und zuklappen. Die Entwickler von NatTable setzen bei die-

Listing 4

```
...
//create row group model
RowGroupModel rowGroupModel = new RowGroupModel<PersonWithAddress>();
rowGroupModel.setDataProvider(bodyDataProvider);

// Create a group of rows for the model.
RowGroup<PersonWithAddress> rowGroup =
  new RowGroup<PersonWithAddress>(rowGroupModel, "Group 1");
                              rowGroup.addMemberRow(bodyDataProvider.getRowObject(0));
rowGroup.addMemberRow(bodyDataProvider.getRowObject(1));
rowGroup.addStaticMemberRow(bodyDataProvider.getRowObject(2));
rowGroupModel.addRowGroup(rowGroup);
...
RowGroupHeaderLayer<PersonWithAddress> rowGroupHeaderLayer =
  new RowGroupHeaderLayer<PersonWithAddress>(
rowHeaderLayer, selectionLayer, rowGroupModel, false);

//configure VerticalTextPainter so row group names are printed vertically
rowGroupHeaderLayer.addConfiguration(
  new DefaultRowGroupHeaderLayerConfiguration<PersonWithAddress>(rowGroupModel) {
    @Override
    public void configureRegistry(IConfigRegistry configRegistry) {
      configRegistry.registerConfigAttribute(
      CellConfigAttributes.CELL_PAINTER,
        new BeveledBorderDecorator(
        new RowGroupHeaderTextPainter<PersonWithAddress>(
        rowGroupModel, new VerticalTextPainter())),
        DisplayMode.NORMAL,
        GridRegion.ROW_GROUP_HEADER
      );
    }
});

rowGroupHeaderLayer.setColumnWidth(20);
...
```

Listing 5

```
//create row group model
RowGroupModel<PersonWithAddress> rowGroupModel =
  new RowGroupModel<PersonWithAddress>();
rowGroupModel.setDataProvider(bodyDataProvider);

// Create a group of rows for the model.
RowGroup<PersonWithAddress> rowGroup =
  new RowGroup<PersonWithAddress>(rowGroupModel, “Group 1”);
rowGroup.addMemberRow(bodyDataProvider.getRowObject(0));
rowGroup.addMemberRow(bodyDataProvider.getRowObject(1));
rowGroup.addStaticMemberRow(bodyDataProvider.getRowObject(2));
rowGroupModel.addRowGroup(rowGroup);
...
RowHideShowLayer rowHideShowLayer =
  new RowHideShowLayer(columnGroupExpandCollapseLayer);
RowGroupExpandCollapseLayer rowGroupExpandCollapseLayer =
  new RowGroupExpandCollapseLayer(rowHideShowLayer, rowGroupModel);
...
```

Listing 6

```
...
treeList = new TreeList<PersonWithAddress>(sortedList,
  new PersonWithAddressTreeFormat(), new PersonWithAddressExpansionModel());
...
GlazedListTreeData<PersonWithAddress> treeData =
                                    new GlazedListTreeData<PersonWithAddress>(treeList) {
  @Override
  public String formatDataForDepth(int depth, PersonWithAddress object) {
    return object.getLastName();
  }
};

TreeLayer treeLayer = new TreeLayer(selectionLayer,
                          new GlazedListTreeRowModel<PersonWithAddress>(treeData), true);
...

private class PersonWithAddressTreeFormat implements TreeList.
                                                        Format<PersonWithAddress> {

private Map<String, PersonWithAddress> parentMapping = new HashMap<String,
                                                                PersonWithAddress>();

public void getPath(List<PersonWithAddress> path, PersonWithAddress element) {
    if (parentMapping.get(element.getLastName()) != null) {
      path.add(parentMapping.get(element.getLastName()));
    } else {
      parentMapping.put(element.getLastName(), element);
    }
    path.add(element);
  }

  public boolean allowsChildren(PersonWithAddress element) {
    return true;
  }

  public Comparator<? extends PersonWithAddress> getComparator(int depth) {
    return new Comparator<PersonWithAddress>() {

    @Override
    public int compare(PersonWithAddress o1, PersonWithAddress o2) {
      return o1.getLastName().compareTo(o2.getLastName());
    }
    };
  }
}

private class PersonWithAddressExpansionModel implements TreeList.
                                                  ExpansionModel<PersonWithAddress> {

  public boolean isExpanded(PersonWithAddress element, List<PersonWithAddress> path) {
    return true;
  }

  public void setExpanded(PersonWithAddress element,  List<PersonWithAddress> path,
                                                                boolean expanded) {
  }
}
```

sem Feature wie bei der Filterzeile auf die Funktionen der *GlazedLists* auf. Sie enthalten die *TreeList*, die über das *TreeList.Format* und das *TreeList.ExpansionModel* konfiguriert wird (Listing 6).

Während das *ExpansionModel* lediglich dafür benötigt wird, auf das Auf-/Zuklappen zu reagieren beziehungsweise den initialen Zustand zurückzuliefern, nachdem ein Element der *TreeList* hinzugefügt wurde, wird das *Format* benötigt, um die Baumstruktur zu definieren. Das schließt die Überprüfung ein, ob Kindknoten erlaubt sind, den *Comparator*, um die Daten in eine sinnvolle Reihenfolge für die Gruppierung bringen zu können, und die Funktion für die Erstellung des Pfades, um die Knoten zu befüllen. Letzteres muss in der Methode *getPath()* ausimplementiert werden. Wichtig zu wissen ist hierbei, dass die übergebene Liste mit den Objekten gefüllt werden muss, die von der Wurzel bis zum Objekt, für das der Pfad angefragt wurde, führen. Es muss mindestens das Objekt der übergebenen Liste hinzugefügt werden, für das der Pfad angefragt wird.

Damit NatTable die Baumstruktur darstellen kann, muss *TreeList* in ein *GlazedListTreeData*-Objekt gewrappt werden. Sinnvollerweise erzeugt man eine eigene, von *GlazedListTreeData* abgeleitete Klasse, in der die Methode *formatDataForDepth()* überschrieben wird, um den Wert, der in der Gruppierungsspalte angezeigt werden soll, korrekt darzustellen. Mit diesem *GlazedListTreeData*-Objekt muss ein *GlazedListTreeRowModel* erzeugt werden, das für die Erstellung des *TreeLayer* benötigt wird. Der *TreeLayer* wird im Body auf den *SelectionLayer* eingefügt.

Wie in **Abbildung 4** zu sehen ist, wurde durch die in Listing 6 aufgeführte Konfiguration eine Tabelle mit Baumstruktur erzeugt, in der der Elternknoten selbst ein vollständig gültiger Eintrag ist. Möchte man statt diesem Verhalten das für Bäume recht typische Verhalten erzeugen, dass der Elternknoten nur eine Gruppierungszeile ist, anstelle eines kompletten Datensatzes, so müssen in der Methode *getPath()* der eigenen *TreeList.Format*-Ausprägung die Gruppierungszeilen entsprechend hinzugefügt werden. Hierbei ist zu beachten, dass entweder Objekte des gleichen Typs mit ansonsten leeren Inhalten für die restlichen Spalten der Tabelle erzeugt und hinzugefügt oder aber Objekte eines anderen Typs hinzugefügt werden. Bei letzterer Methode muss darauf geachtet werden, dass für alle Klassen, die anhand des definierten Tabellendatentyps agieren, zum Beispiel des eigenen *IConfigLabelAccumulator* oder *ColumnPropertyAccessor*, Typprüfungen durchgeführt werden müssen, da durch diese Anpassung eine Liste mit unterschiedlichen Datentypen erzeugt wird. Die nachfolgend beschriebene *GroupBy*-Funktion nutzt genau diesen Mechanismus intern, um die Gruppierungszeilen zu erzeugen.

Group By

Seit der Version 2.3.0 bietet NatTable mithilfe der *Tree*-Funktion das *GroupBy*-Feature an. Es ermöglicht dem

Benutzer, die Tabelle anhand von Spalten zu gruppieren, wofür er lediglich die Spalte per Drag and Drop in den *GroupBy*-Bereich der Tabelle ziehen muss. Anhand der ausgewählten Spalte wird in der ersten Spalte der Tabelle mithilfe des *Tree*-Konzepts eine Gruppierungszeile eingefügt, unter der die zugehörigen Datenzeilen auf- und zugeklappt werden können. Es ist möglich, beliebig viele Spalten in den *GroupBy*-Bereich zu ziehen, um komplexe Gruppierungen zu erstellen (**Abb. 5**). Die für die Gruppierung verwendeten Spalten werden im *GroupBy*-Bereich dargestellt und können per Rechtsklick wieder entfernt werden.

Um die *GroupBy*-Funktion zu verwenden, müssen im Gegensatz zum eigenen Tree nur wenige Schritte durchgeführt werden. Viele der beschriebenen Schritte, die notwendig sind, um selbst einen Baum in NatTable einzubauen, wurden in den *GroupBy*-Klassen gekapselt und verallgemeinert, sodass es keiner speziellen Implementierung bedarf.

	Lastname	Firstname	Gender	Married	Birthday	Street	Housenumber	Postal Code	City
1	Carlson	Timothy	MALE	☐	Sat Nov 11 00:00:...	Plympton Street	169	55555	North Haverbrook
2	⊞ Flanders	Lisa	FEMALE	☐	Mon Jul 09 00:00:...	Elm Street	97	66666	Capital City
3	⊞ Krabappel	Edna	FEMALE	☐	Sun Sep 23 00:00...	South Street	168	44444	Waverly Hills
4	⊞ Leonard	Carl	MALE	☐	Wed Jul 05 00:00...	Elm Street	130	66666	Capital City
5	⊟ Lovejoy	Lisa	FEMALE	☑	Sun Jul 28 00:00:...	Main Street	47	33333	Ogdenville
6	Lovejoy	Carl	MALE	☑	Tue Oct 10 00:00...	Evergreen Terrace	71	66666	Capital City
7	Lovejoy	Homer	MALE	☑	Wed Oct 27 00:0...	Plympton Street	178	33333	Ogdenville
8	Lovejoy	Carl	MALE	☑	Sat Aug 03 00:00...	South Street	156	66666	Capital City
9	⊟ Simpson	Jessica	FEMALE	☑	Sat Apr 24 00:00:...	Plympton Street	185	33333	Ogdenville
10	Simpson	Homer	MALE	☐	Sat Sep 23 00:00...	Oak Grove Street	94	11111	Springfield
11	Simpson	Bart	MALE	☑	Sun Dec 31 00:00...	Elm Street	74	55555	North Haverbrook
12	⊟ Smithers	Bart	MALE	☑	Tue Mar 16 00:00...	Highland Avenue	119	11111	Springfield
13	Smithers	Homer	MALE	☐	Fri May 22 00:00:...	Main Street	43	22222	Shelbyville
14	Smithers	Edna	FEMALE	☐	Mon Sep 22 00:0...	Oak Grove Street	193	44444	Waverly Hills

Abb. 4: NatTable mit Verwendung von Trees (TreeNatTableExample.java)

Lastname › Firstname | Ungroup By

	Lastname	Firstname	Gender	Married	Birthday	Street	Housenumber	Postal Code	City
127	⊟ Simpson								
128	⊟ Bart								
129	Simpson	Bart	MALE	☑	Wed Apr 05 00:0...	Elm Street	27	11111	Springfield
130	Simpson	Bart	MALE	☐	Fri Aug 07 00:00:...	Evergreen Terrace	177	33333	Ogdenville
131	⊟ Carl								
132	Simpson	Carl	MALE	☑	Fri Jan 12 00:00:...	Plympton Street	98	55555	North Haverbrook
133	⊟ Helen								
134	Simpson	Helen	FEMALE	☐	Mon Jun 21 00:00...	Highland Avenue	43	66666	Capital City
135	Simpson	Helen	FEMALE	☑	Tue May 02 00:0...	Elm Street	54	66666	Capital City
136	⊟ Homer								
137	Simpson	Homer	MALE	☑	Sun Jul 05 00:00:...	Elm Street	120	33333	Ogdenville
138	⊟ Jessica								
139	Simpson	Jessica	FEMALE	☑	Fri Apr 13 00:00:...	Plympton Street	148	55555	North Haverbrook
140	⊟ Lenny								
141	Simpson	Lenny	MALE	☑	Sat Feb 16 00:00...	Evergreen Terrace	49	33333	Ogdenville
142	⊟ Maggie								
143	Simpson	Maggie	FEMALE	☐	Mon Feb 27 00:0...	Evergreen Terrace	102	22222	Shelbyville
144	⊟ Marge								
145	Simpson	Marge	FEMALE	☑	Wed Dec 05 00:0...	Elm Street	99	33333	Ogdenville
146	⊟ Waylon								
147	Simpson	Waylon	MALE	☐	Thu Oct 07 00:00...	Elm Street	1	22222	Shelbyville
148	⊟ Smithers								
149	⊟ Carl								
150	Smithers	Carl	MALE	☐	Thu Oct 13 00:00...	Oak Grove Street	24	44444	Waverly Hills

Abb. 5: GroupBy in NatTable (GroupByNatTableExample.java)

Statt des normalen *DataLayer* muss der spezielle *GroupByDataLayer* im Body verwendet werden. Der fertig aufgebaute *GridLayer* muss anschließend noch wie in Listing 7 zusammen mit dem *GroupByHeaderLayer* in einem *CompositeLayer* zusammen gepackt und auf NatTable gesetzt werden. Außerdem sollte auf NatTable noch die *GroupByHeaderMenuConfiguration* gesetzt werden, damit die Gruppierungen über Rechtsklick im *GroupBy*-Bereich wieder aufgelöst werden können.

Da die *GroupBy*-Funktion die Trees verwendet, muss auch hier beim *IColumnPropertyAccessor* darauf geachtet werden, dass entsprechende Typprüfungen durchgeführt werden, da die Gruppierungszeilen Objekte vom Typ *GroupByObject* darstellen.

Actions, Commands, Events

Benutzerinteraktionen werden im Zusammenhang mit NatTable über Actions, Commands und Events verarbeitet. Während Actions über die *UiBindingRegistry* direkt an entsprechende Benutzerinteraktionen gebunden werden, können Commands über NatTable ausgeführt und Events über Layer gefeuert werden. Eigene Key und Mouse Bindings können über die *UiBindingRegistry* eingebunden werden. Hierzu muss, je nachdem, was für eine Action überschrieben werden soll, eines der drei Interfaces *IKeyAction*, *IMouseAction* oder *IDragMode* und die jeweilige *run()*-Methode implementiert werden. Die Action kann dann über die entsprechende *registerXxx*-Methode der *UiBindingRegistry* an Benutzerinteraktionen via *MouseEventMatcher* beziehungsweise *KeyEventMatcher* gebunden werden.

In der Regel lösen Actions die Ausführung von Commands aus, da sie im NatTable-Konzept für die Ausführung von Funktionen verwendet werden. Auch für die programmatische Ausführung von Funktionen können

Listing 7

```
...
CompositeLayer compositeGridLayer = new CompositeLayer(1, 2);
GroupByHeaderLayer groupByHeaderLayer =
  new GroupByHeaderLayer(grid.getGroupByModel(), grid,
    grid.getColumnHeaderDataLayer().getDataProvider());
compositeGridLayer.setChildLayer(GroupByHeaderLayer.GROUP_BY_REGION,
  groupByHeaderLayer, 0, 0);
compositeGridLayer.setChildLayer("Grid", grid, 0, 1);

NatTable natTable = new NatTable(parent, compositeGridLayer, false);
natTable.addConfiguration(new GroupByHeaderMenuConfiguration(natTable,
                                                    groupByHeaderLayer));
...
```

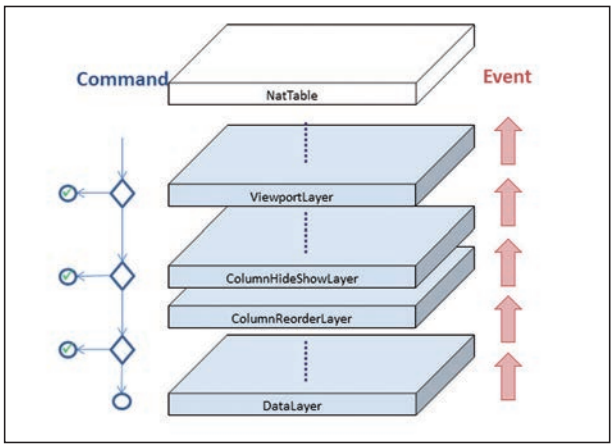

Abb. 6: Verarbeitung von Commands und Events in NatTable

Commands verwendet werden. Ein Command ist eine von *ILayerCommand* abgeleitete Klasse, die die Informationen für die Verarbeitung der Funktion enthält. Es kann via *doCommand()* auf NatTable oder einem beliebigen Layer ausgeführt werden. Dabei wird das *ILayerCommand* den Layer Stack abwärts weitergereicht, bis es von einem *ILayer* oder einem *ILayerCommandHandler* verarbeitet wird. Um ein eigenes *ILayerCommand* zu erstellen, kann eine der vielen vorbereiteten abstrakten Implementierungen erweitert werden. Wichtig zu wissen ist dabei, dass die Methode *cloneCommand()* implementiert werden muss, um das Weiterreichen des Commands im Layer Stack zu unterstützen, während die Methode *convertToTargetLayer()* implementiert werden muss, um die korrekte Zeilen- und Spaltenposition für den entsprechenden Layer zu ermitteln. Die Verarbeitung eines *ILayerCommand* kann sowohl von einem Layer selbst übernommen werden als auch von einem *ILayerCommandHandler*, der auf den Layer gesetzt werden muss. Im Sinne der Kapselung und der Wiederverwendbarkeit ist die Implementierung eines *ILayerCommandHandler* vorzuziehen, da die Verarbeitung von neuen *ILayerCommands* auch über bestehende Layer durchgeführt werden kann.

Der *ILayerCommandHandler* muss über die Methode *getCommandClass()* Auskunft darüber liefern können, für welche Art von *ILayerCommands* er zuständig ist. Das ist notwendig, um feststellen zu können, wann das zu verarbeitende *ILayerCommand* verarbeitet werden kann. Die Verarbeitung selbst wird in der Methode *doCommand()* implementiert, die *true* zurückliefert, wenn das *ILayerCommand* verarbeitet werden konnte. Liefert die Methode stattdessen *false* zurück, so wird das *ILayerCommand* weiterhin im Layer Stack abwärts weitergereicht, bis es verarbeitet werden konnte. Der *ILayerCommandHandler* muss via *registerCommandHandler()* auf dem Layer registriert werden, auf dem die Ausführung des *ILayerCommands* gewünscht ist, beispielsweise werden die *ILayerCommandHandler* für Selektionen auf dem *SelectionLayer* registriert.

Während Commands bis zum verarbeitenden Layer den Layer Stack abwärts propagiert werden, werden Events den Layer Stack aufwärts verbreitet (**Abb. 6**). Entgegen der Verarbeitung von Commands werden Events allerdings auch nach der Verarbeitung weitergereicht, um jedem Layer, der das *ILayerListener*-Interface implementiert, die Möglichkeit zu geben, auf das Event zu reagieren. *ILayerEvents* werden vor allem dazu verwendet, nach der Ausführung eines *ILayerCommands* jeden Layer über Änderungen zu informieren. Möchte man ein eigenes *ILayerEvent* implementieren, müssen wie für das *ILayerCommand* Methoden für das Klonen und das Konvertieren der Zeilen-/Spaltenpositionen implementiert werden. Für den Normalfall sollten allerdings die zahlreich vorhandenen *ILayerEvents* von NatTable ausreichen. Am häufigsten dürften die von *IStructuralChangeEvent* abgeleiteten Events wie das generelle *StructuralRefreshEvent* im Eigengebrauch zum Einsatz kommen.

In der Regel werden Actions, Commands und Events zusammen verarbeitet. Beispielsweise wird über einen Mausklick eine Action angestoßen, die wiederum ein Command startet, das nach seiner Ausführung über ein Event den Refresh auf NatTable veranlasst. Das *CommandNatTableExample* auf der Heft-CD zeigt ein Beispiel für die Erstellung und Einbindung eigener Actions und Commands.

Notwendige Plug-ins

Um NatTable mit allen Funktionen verwenden zu können, benötigen Sie folgende Libraries:

- glazedlists_java15-1.8.0.jar
- net.sourceforge.nattable.core-2.3.1.1.jar
- net.sourceforge.nattable.extension.glazedlists-2.3.1.1.jar
- net.sourceforge.nattable.extension.poi-2.3.1.1-.jar
- org.apache.poi-3.7.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. Die genannten JARs finden Sie online unter [3], [4], [5] oder auf der Heft-CD.

Dirk Fauth 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 aktiver Committer im NatTable-Projekt.

Links & Literatur

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

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

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

[4] http://sourceforge.net/projects/nattable/files/NatTable/

[5] http://sourceforge.net/projects/nattable/files/org.apache.poi/

[6] http://sourceforge.net/projects/nattable/forums/forum/744992/topic/4726129