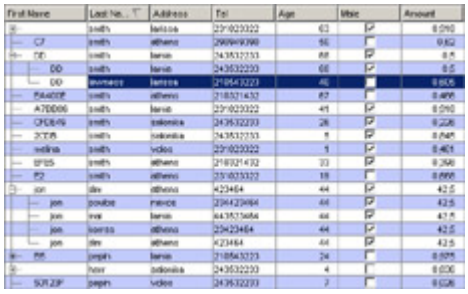

Citra Table 4.0
Developer's Manual

version 2.0

Table of Contents

1 Preface

One of the most commonly used Java GUI widgets is the JTable. JTable's framework is extremely flexible and powerful, yet lacks the functionality which is often required in modern applications. By providing an extension to the JTable framework, our table component attempts to make these missing features available to Java developers, without sacrificing simplicity or performance. Moreover, the design of the library follows a minimalistic approach, featuring a small hierarchy tree, which makes it fast and easy to understand and incorporate into existing code.

The purpose of this document is to make developers familiar with Citra Table and to explain its main aspects. Please note that this guide is NOT intended to be exhaustive or complete.

2 TableModels

In JTable's framework, **a TableModel** contains the data of the table. JTable interrogates this data model so that it can paint itself accordingly. By employing the same mechanism and by extending the TableModel interface, we managed to provide effects, such as sorting and filtering, without the requirement of a special JTable subclass.

2.1 ListTableModel

We extended the TableModel interface with **ListTableModel**, that requires the tabular data to be kept in a list structure. ListTableModel is the interface that is inherited by other TableModels in our library to provide effects such as sorting (SortTableModel), filtering (FilterTableModel) or tree-like viewing (TreeTableModel). ListTableModel extends the TableModel interface and also defines methods for manipulating the data of a tabular data model.

By implementing ListTableModel in your TableModel, it is assumed that the data are stored in a collection indexed by row number. This collection is returned with the method:

public java.util.List getRows();

Hence, if we want to get the object at the 1st row:

Object rowObject = ListTableModel.getRows().get(0);

Some additional methods are used to add and remove elements in the list. These are:

public void addRow(Object row);
public void addRows(List addedRows);
public void clear();
public void removeRow(int row);

public void removeRows(int [] deletedRows);

The elements in the list represent the actual row data, which can be any Java object. To get a reference to a specific column of a row data, use:

public Object getCellValue(Object row, int index);

For example, for a DefaultTableModel subclass implementing ListTableModel, the method's body would be:

```
public Object getCellValue(Object row, int index) {
        return ((Vector) row).get(index);
};
```

The TableModels in the library (SortTableModel, FilterTableModel and TreeTableModel) all require a ListTableModel in their constructor. Therefore, in order to use your own custom TableModel in conjunction with these classes, your TableModel should implement the ListTableModel interface. This can be very easily done. Please look in Appendix I, where the source code listing for a ListTableModel implementation that extends javax.swing.table.DefaultTableModel is presented.

2.2 ObjectTableModel

ObjectTableModel is a **ListTableModel** whose data is a collection of arbitrary Java Objects. This abstract class uses an internal ArrayList structure to store the rows of a JTable. Subclasses should implement the methods:

public Object getValueAt(Object rowObject, int index);
public void setValueAt(Object rowObject, Object aValue, int index);

, which define how objects at a column index are retrieved/set respectively.
These methods should preferrably cast the supplied Object argument to an appropriate class type, and use the index parameter to get to the actual cell value.

Also, the column names are specified in the constructor, or with the method:

public void setColumns(String[] columnNames);

Example: Make an ObjectTableModel for the object Employee given below:

```
public class Employee {
        public String firstName;
        public String lastName;
        public Integer age;
        Employee(String firstName, String lastName, int age) {
                this.firstName = firstName;
```

```
                this.lastName = lastName;
                this.age = new Integer(age);
        }
}

public class IDTableModel exends ObjectTableModel {
        public IDTableModel() {
                super(new String[]{"First Name", "Last Name", "Age"});
        }
        public Object getValueAt(Object o, int index) {
                Employee emp = (Employee) o;
                Object ret = null;
                switch (index) {
                        case 0: {
                                ret = emp.firstName;
                                break;
                        }
                        case 1: {
                                ret = emp.lastName;
                                break;
                        }
                        case 2: {
                                ret = emp.age;
                                break;
                        }
                }
                return ret;
        }
        public void setValueAt(Object o, Object aValue, int index) {
                Employee emp = (Employee) o;
                switch (index) {
                        case 0: {
                                emp.firstName = value.toString();
                                break;
                        }
                        case 1: {
                                lastName = value.toString();
                                break;
                        }
                        case 2: {
                                emp.age = Integer.valueOf(value.toString());
                                break;
                        }
                }
        }
}
```

2.3 ListTableMap

ListTableMap defines a TableModel that wraps around a **ListTableModel**, which is passed as an argument in the constructor. The methods of ListTableModel that are implemented are nothing more than calls to the respective methods of the underlying ListTableModel. In addition, events generated from the underlying ListTableModel are intercepted and sent to ListTableMap's table model listeners.

Other TableModels in the library extend ListTableMap, such as SortTableModel, FilterTableModel, TreeTableModel. All these ListTableMap's subclasses transform the underlying data in some way. By **chaining** multiple ListTableMaps, a cumulative transformation is achieved.

Example: Given the IDTableModel created in the previous section, make a sortable and filterable data model.

```
//create the chain of TableModels
IDTableModel originalModel = new IDTableModel();
FilterTableModel ftm = new FilterTableModel(originalModel);
SortTableModel stm = new SortTableModel(ftm);

//set the data model of the table to the last chained TableModel created.
JTable table = new JTable(stm);
OR
JTable table = new AdvancedJTable(stm);
```

3 AdvancedJTable

AdvancedJTable extends JTable, so it inherits all JTable's methods and properties.

In addition, it provides a rich set of features made available by other classes in the library. More specifically, AdvancedJTable can:

- Autoresize table columns upon double-clicking on a table column border.
- Display a popup through which the columns of the table can be dynamically added/removed.
- Use a table header whose columns cannot be reordered with right mouse button clicks. You can also specify which columns are allowed to be dragged and reordered.
- Correctly handle selection changes when the tabular data are restructured.
- Merge and split cells.
- Display a row header.
- Group column headers together.
- Lock non-scrollable rows/columns on the edges of the table.
- Use Styles for easy cell rendering.

- Use additional cell editors and renderers.

Furthermore, AdvancedJTable includes some inherent features:

- A dummy column may be added last to the table header with no data underneath, for decorative purposes. See Dummy Last Column.
- Make the first column of the table not movable. See Fixed First Column.
- Programmatically alter the table columns displayed and their widths. See Table State.

Finally, by using specialized TableModels, AdvancedJTable manipulates the data in order to provide for sorting and filtering effects.

3.1 Creating

AdvancedJTable has exactly the same constructors as JTable. You can therefore construct an instance the same way you do with JTable.

Example:

```
//construct an AdvancedJTable with 5 rows and 3 columns
AdvancedJTable table = new AdvancedJTable(5, 3);

//construct an AdvancedJTable with a DefaultTableModel as its datamodel.
DefaultTableModel dtm = new DefaultTableModel();
AdvancedJTable table = new AdvancedJTable(dtm);

//construct an AdvancedJTable with a couple of rows.
Object[][] rowData = new String[][] {
{"Mary", "Lloyds"},
{"John", "Berry"}
};
Object[] columns = new String[]{"Name", "Surname"};
DefaultTableModel model = new DefaultTableModel(rowData, columns);
AdvancedJTable table = new AdvancedJTable(model);
```

3.2 Inherent Features

These features exist only in AdvancedJTable and its subclasses.

3.2.1 Dummy Last Column

There is the option to add a dummy column to the table header with no data underneath, for decorative purposes.

You can control the visibility of the dummy column with:

public void setShowDummyColumn(boolean showDummyColumn);

So,

```
AdvancedJTable table = new AdvancedJTable();
table.setShowDummyColumn(true);
```

This option is available ONLY if the table is enclosed by an AdvancedJScrollPane:

```
AdvancedJTable table = new AdvancedJTable();
table.setShowDummyColumn(true);
AdvancedJScrollPane scroller = new AdvancedJScrollPane();
scroller.setViewportView(table);
```

Finally, you can determine if the dummy column is present by calling:

public boolean getShowDummyColumn();

OR

public boolean isDummyColumn(int column);

3.2.2 Fixed First Column

You can make the first column of the table non-reorderable and irremovable by calling AdvancedJTable's method:

public void setFirstColumnFixed(boolean isFirstColumnFixed);

So,

```
AdvancedJTable table = new AdvancedJTable();
table.setFirstColumnFixed(true);
```

The first column will now be locked in place.
If you try to drag another column over to the first column, your dragging will be cancelled without moving the first column.

You can also query the state of the first column by calling:

public boolean isFirstColumnFixed();

Note: In order to make the first column non-reorderable and irremovable, AdvancedJTable uses subclasses of DefaultTableColumnModel (AdvancedJTable.InnerTableColumnModel) and JTableHeader (AdvancedJTable.InnerTableHeader) as its table column model and header respectively. Therefore, if you would like to use your own table column model or header, your classes must extend AdvancedJTable's inner classes mentioned above.

3.2.3 Table State

You can programmatically alter the table columns displayed and their widths by using:

public void setTableState(String state);

The string to pass in this method should be specially formatted as follows:

<state> = i_1:w1,i2:w2,... where ***i*** is the column's model index and ***w*** its width.

You can also get a string of the table's state by calling:

public String getTableState();

Note: A column which is removed by calling this method can always be added later if its model index is supplied. Only columns that are removed via the removeColumn method of JTable are permanently removed from the table.

3.3 Common Features

The features described here are named ***common*** because they can be made available to JTable's subclasses, other than AdvancedJTable. These features are described extensively in their own section.

3.3.1 Autoresize Table Columns

You can automatically resize the column of a table to the greatest preferred width of all cells under that column, when the column is double-clicked on its border.
See Table Assistant and Autoresize Table Columns for more information.

3.3.2 Column Filter

You can filter which columns to appear on the table through a popup menu.
See Table Assistant and Column Filter for more information.

3.3.3 Reorderable Column Header

AdvancedTable uses a table header that does not let column reordering when the column is being dragged with the right mouse button pressed. Additionally, you can specify which columns are allowed to be dragged and reordered.
See AdvancedTableHeader for more information.

3.3.4 Table Selections after data change

Row selections are cleared after a data change event. AdvancedJTable is able to restore the table's row selection after a data change.
See TableReorder for more information.

3.3.5 Cell Spanning

AdvancedJTable is able to merge and split cells.
See Cell Spanning for more information.

3.3.6 Row Header

AdvancedJTable provides a row header.
See JTableRowHeader for more information.

3.3.7 Groupable Header

AdvancedJTable's column header is a GroupTableHeader which provides for groupable column headers.
See GroupTableHeader for more information.

3.3.8 Locked Rows/Columns

You can lock non-scrollable rows/columns on the edges of AdvancedJTable by using a LockedModel.
See the more general Locked Rows/Columns chapter for more information.

3.3.9 Visual Appearance

AdvancedJTable uses specialized Objects for modifying its visual appearance.
See Styles for more information.

3.3.10 Editors

AdvancedJTable installs a DateEditor for editing Date values.
See Editors and Date Editor for more information.

4 TreeTable (old impl.)

WARNING: This chapter refers to the original TreeTable implementation, which is now **deprecated**. In version 3.0, we introduced a more flexible TreeTable framework for dealing with tree-tables in Java, which is contained in the **com.citra.treetable** package. You are encouraged to use this new TreeTable and accompanying classes. We tried, where possible to preserve the class names, so as to facilitate the transition. Due to this fact, one must be careful not to mix classes from the **com.citra.tree package** with classes from the **com.citra.treetable** package.

TreeTable is the combination of a tree and a table. You can use this component in order to group several rows of a table in a single row. TreeTable extends **AdvancedJTable** and therefore inherits all of its methods and properties. See AdvancedJTable for more information. While TreeTable makes it possible to have a tree inside a table, a **TreeTableModel** is the accompanying table model that creates and maintains a dynamic tree structure. This structure is determined by the underlying data and by special objects, called **TreeTableComparators**, that define how the grouping of rows should be performed. Finally, the branch nodes (nodes that have children) of the tree structure are called aggregate rows, and their cell values on the table are calculated with the help of **Aggregators**.

4.1 Creating

TreeTable uses a TreeTableModel as its table model, which can be specified at construction time using the constructor:

public TreeTable(TreeTableModel model);

You can also create a TreeTable by specifying the underlying data with a ListTableModel:

public TreeTable(ListTableModel model);

In this case, a TreeTableModel will be created automatically with the method:

protected static TreeTableModel createDefaultTreeTableModel(ListTableModel tableModel);

The TreeTableModel created will provide a wrapping around the supplied ListTableModel, and will also become the TreeTable's data model.

4.2 TreeTableModel

WARNING: Since version 3.0, where we introduced a new TreeTable implementation, TreeTableModel has been replaced with DynamicTreeTableModel.

TreeTableModel is TreeTable's data model. Setting a model other than this, will result in an exception being thrown. TreeTableModel is a **ListTableMap** that wraps around a ListTableModel instance. When the data of the underlying model changes, TreeTableModel dynamically builds a tree, whose structure depends on the actual data and on **TreeTableComparators**, that define the way the grouping of rows should be performed.

4.2.1 Creating

By extending **ListTableMap**, TreeTableModel wraps around a **ListTableModel** that provides the actual data. The ListTableModel to use is specified in the sole constructor:

public TreeTableModel(ListTableModel tableModel);

You can also set the ListTableModel instance later using ListTableMap's method:

public void setModel(ListTableModel newModel);

Example 1: Create a TreeTableModel and set it to a treetable.

```
//create the table models.
ListTableModel flatModel = new DefaultListTableModel();
TreeTableModel ttm = new TreeTableModel(flatModel);

//create the table
TreeTable table = new TreeTable(ttm);
```

Example 2: Let the TreeTable to create a TreeTableModel for us.

```
//create the flat table model.
ListTableModel flatModel = new DefaultListTableModel();

//create the table
TreeTable table = new TreeTable(flatModel);

//get an instance to the implicitly created TreeTableModel
TreeTableModel ttm = (TreeTableModel) table.getModel();
```

Example 3: Create a sortable and filterable TreeTableModel and set it to a treetable.

```
//create the chain of table models.
ListTableModel flatModel = new DefaultListTableModel();
FilterTableModel ftm = new FilterTableModel(flatModel);
SortTableModel stm = new SortTableModel(ftm);
TreeTableModel ttm = new TreeTableModel(stm);

//create the table
TreeTable table = new TreeTable(ttm);

//take care of the SortTableModel's header renderer
stm.setHeader(table.getTableHeader());
```

4.2.2 TreeTableRows

The nodes of the tree structure are instances of the abstract class **TreeTableRow**, which extends **DefaultMutableTreeNode**. They are divided in:

1. **DataRows**: nodes that are associated with the actual data of the underlying ListTableModel. These nodes cannot have children.
2. **AggregateRows**: nodes that are not associated with the data of the underlying ListTableModel, but that may provide information about several rows of the table. These are further classifed into:
   - **HeaderRows**: branch nodes that have children and that can be expanded. HeaderRows usually contain information about their children.
   - **FooterRows**: nodes placed at the bottom of each tree level. FooterRows usually contain information about the rows above them. They are added to the tree as long as a **Footer** is defined in the TreeTableModel.

A TreeTableRow is constructed by specifying the data row object and the index that corresponds to that object in the data list of the underlying ListTableModel:

```
public TreeTableRow(Object o, int modelIndex);
```

TreeTableRow also has methods that determine its type:

```
public boolean isAggregate();
public boolean isHeader();
public boolean isFooter();
```

Finally, AggregateRow defines methods for getting and setting the aggregate values:

```
public Object getAggregateValue(int rowIndex, int columnIndex);
public void setAggregateValue(Object value, int rowIndex, int columnIndex);
```

4.2.3 Getting to the data

TreeTableModel builds the tree structure dynamically every time the underlying data changes. To get from the "tree-view" to the original data model provided by the underlying ListTableModel, you can use the following method:

public int getDataRow(int rowIndex);

The above method assumes that **rowIndex** is a DataRow (single). In order to retrieve the indexes for HeaderRows, use:

public int[] getDataRows(int rowIndex);
public int[] getModelIndexesUnderRow(int row, boolean sorted);
public int[] getModelIndexesUnderRow(TreeTableRow row, boolean sorted);

Furthermore, there are methods for examining the nodes of the tree - the TreeTableRows:

public TreeTableRow getTreeRow(int rowIndex);
public int getLevel(int rowIndex);
public boolean isAggregate(int rowIndex);
public boolean isFooter(int rowIndex);
public boolean isHeader(int rowIndex);

Example 1: Find the objects that correspond to a treetable's row selection. (using the user object of the tree node)

```
//table is the TreeTable model
int[] selectedRows = table.getSelectedRows();

TreeTableModel ttm = (TreeTableModel) table.getModel();
List treeList = ttm.getRows();
for (int i=0;i<selectedRows.length;i++) {
        TreeTableRow treeRow = (TreeTableRow) treeList.get(selectedRows[i]);
        Object objectRow = treeRow.getUserObject();
}
```

Example 2: Find the objects that correspond to a treetable's row selection. (using the underlying ListTableModel)

```
//table is the TreeTable model
int[] selectedRows = table.getSelectedRows();

TreeTableModel ttm = (TreeTableModel) table.getModel();
ListTableModel wrappedModel = ttm.getModel();
for (int i=0;i<selectedRows.length;i++) {
        int origIndex = ttm.getDataRow(selectedRows[i]);
        Onject objectRow = wrappedModel.getRows().get(origIndex);
```

}

4.2.4 TreeTableComparators

TreeTableComparators are used in order to dynamically group the rows of a TreeTable component. TreeTableComparator implements the java.util.Comparator interface, thus the method:

public int compare(Object o1, Object o2);

, should be implemented.

The supplied objects to the **compare** method above are the row data objects of the underlying ListTableModel.

TreeTableModel manages a collection of TreeTableComparators. The collection can be manipulated with the methods:

public void addRowComparator(TreeTableComparator newComparator);
public void insertRowComparator(TreeTableComparator newComparator, int index);
public TreeTableComparator removeRowComparator(int index);
public boolean removeRowComparator(TreeTableComparator comparator);
public TreeTableComparator setRowComparator(TreeTableComparator newComparator, int index);
public TreeTableComparator[] getRowComparators();
public TreeTableComparator getRowComparator(int index);

A TreeTableComparator implementation, **DefaultTreeTableComparator**, compares row data based on a single column. The comparator to use for the column is retrieved with TreeTableModel's method:

public Comparator getDefaultComparator(Class columnClass);

Also, TreeTableModel installs comparators for all the common classes with the method:

protected void createDefaultComparators();

You can assign the comparator to use for a column with:

public void setDefaultComparator(Class columnClass, Comparator comparator);

It is up to the developer to use the installed 'class' comparators with their own TreeTableComparator implementation.

Example 1: Create a TreeTableComparator that compares data based on the first column

```
//ttm is the TreeTableModel

TreeTableComparator myComparator = new TreeTableComparator(ttm) {
        public int compare(Object o1, Object o2) {
                //get the cell value for column 0
                Object val1 = model.getCellValue(o1, 0);
                Object val2 = model.getCellValue(o2, 0);

                //get a comparator from the treetable model using the object's class
                Comparator c = model.getDefaultComparator(val1.getClass());

                //we can use this comparator to make the comparison
                int comparison = c.compare(val1, val2);

                //return
                return comparison;
        }
        public boolean isGroupedByColumn(int column) {
                return column == 0;
        }
};
```

Example 2: Create a TreeTableComparator that compares data based on the first and the second column

```
//ttm is the TreeTableModel

TreeTableComparator myComparator = new TreeTableComparator(ttm) {
        public int compare(Object o1, Object o2) {
                //get the cell value for column 0
                Object val1 = model.getCellValue(o1, 0);
                Object val2 = model.getCellValue(o2, 0);

                //get the cell value for column 1
                Object val3 = model.getCellValue(o1, 1);
                Object val4 = model.getCellValue(o2, 1);

                //get a comparator for the first column from the treetable model using the
object's class
                Comparator c1 = model.getDefaultComparator(val1.getClass());

                //get a comparator for the second column from the treetable model using
the object's class
                Comparator c2 = model.getDefaultComparator(val3.getClass());

                //we can use these comparators to make the comparison
                int comparison1 = c1.compare(val1, val2);
                int comparsion2 = c2.compare(val3, val4);
```

```
            //return
            return comparison1 == 0 ? comparison2 : comparison1;
        }
        public boolean isGroupedByColumn(int column) {
            return column == 0 && column == 1;
        }
};
```

Example 3: Create a TreeTableComparator that compares data based on the objects supplied. We assume that the supplied objects implement the Comparable interface.

```
//ttm is the TreeTableModel

TreeTableComparator myComparator = new TreeTableComparator(ttm) {
        public int compare(Object o1, Object o2) {
            Comparable c1 = (Comparable) o1;
            return c1.compareTo(o2);
        }
        public boolean isGroupedByColumn(int column) {
            return false;
        }
};
```

4.2.5 Aggregators

Aggregators calculate and return values for the aggregate rows of a TreeTable component. The value for the cell of an aggregate row is calculated with:

```
public Object getAggregateValue(int rowIndex, int columnIndex);
```

Then, the value is assigned on the row with:

```
public Object prepare(AggregateRow row, int rowIndex, int columnIndex);
```

, which ensures that aggregate values are not calculated repeatedly.

TreeTableModel defines methods for creating, assigning and retrieving the aggregators:

```
protected Aggregator createDefaultAggregator();
public void setDefaultAggregator(Aggregator aggregator);
public Aggregator getAggregator(int rowIndex, int columnIndex);
public Aggregator getDefaultAggregator();
```

By default, a DefaultCellAggregator is created which uses the installed **TreeTableComparators** to calculate the aggregate cell values.

4.2.6 Footers

The **Footer** interface defines the place and number of **FooterRows** to add to a TreeTableModel via the method:

public int getFooterSize(TreeTableRow row);

Footer implementations should return the number of footers to add under the supplied TreeTableRow. An appropriate aggregator should also be used to calculate the cell values for these FooterRows.

TreeTableModel defines methods for creating, assigning and retrieving the footer:

```
protected Footer createDefaultFooter();
public void setFooter(Footer footer);
public Footer getFooter();
```

TreeTableModel will use the assigned footer in order to add footer rows to the tree structure. This is accomplished with the method:

protected void buildFooter();

Example: Create and install a custom footer and an accompanying aggregator that sums over the integers values of the cells.

```
//ttm is the TreeTableModel

//create the footer
Footer myFooter = new Footer() {
        public int getFooterSize(TreeTableRow row) {
                if (row.getLevel() == 0) return 0;
                return 1;
        }
};

//create the aggregator
Aggregator footerAggregator = new DefaultCellAggregator(ttm) {
        public Object getAggregateValue(int rowIndex, int columnIndex) {
                if (columnIndex == 4 && model.isFooter(rowIndex)) {
                        TreeTableRow node = model.getTreeRow(rowIndex);
                        TreeTableRow parent = (TreeTableRow) node.getParent();

                        int[] totalRows = model.getModelIndexesUnderRow(parent, false);
                        int sum = 0;
                        for (int i = 0; i < totalRows.length; i++) {
```

```
                                Integer iv = (Integer)
model.getModel().getValueAt(totalRows[i], 4);
                                int ival = iv.intValue();
                                sum += ival;
                        }
                        return new Integer(sum);
                }
                return super.getAggregateValue(rowIndex, columnIndex);
        }
};

//set the footer
ttm.setFooter(myFooter);

//set the aggregator
ttm.setDefaultAggregator(footerAggregator);
```

4.3 Aggregate Renderers

An **AggregateRenderer** is used to render the TreeTable's aggregate rows. The aggregate renderer for a cell is taken with either specifying the row and column index for the cell, or the class of the cell's value:

```
public TableCellRenderer getAggregateCellRenderer(int row, int column);
public TableCellRenderer getDefaultAggregateRenderer(Class columnClass);
```

You can also set the based-on-class renderer with:

```
public void setDefaultAggregateRenderer(Class columnClass, TableCellRenderer
renderer);
```

By default, TreeTable will install **DefaultAggregateRenderer** instances for all the basic classes (String, Object, Date, Boolean and Number).

4.4 Cell Spanning

TreeTable extends AdvancedJTable and is therefore capable of cell spanning. TreeTable overrides AdvancedJTable's method:

```
protected SpanDrawer createSpanDrawer();
```

, in order to set an inner DefaultSpanModel subclass, **TreeTable.DefaultTreeSpanModel**, which spans the header rows of the treetable. Here is how DefaultTreeSpanModel is implemented:

```
public class DefaultTreeSpanModel extends DefaultSpanModel {
        private CellSpan cs;
        /**
         * Constructs a DefaultTreeSpanModel.
         */
        public DefaultTreeSpanModel() {
                cs = new CellSpan(0, 0, 0, CellSpan.ALL_COLUMNS);
        }
        public CellSpan getCellSpanAt(int row, int column) {
                TreeTableModel model = (TreeTableModel) getModel();
                if (model.isHeader(row)) {
                        cs.setSpannedRow(row);
                        return cs;
                }
                return super.getCellSpanAt(row, column);
        }
}
```

4.5 Grouping Panel

GroupingPanel is a panel through which users can dynamically control the structure of a TreeTable. GroupingPanel uses a box layout in order to layout a number of comboboxes, whose items are populated with the columns of a table. By selecting a column in the combo box, the appropriate **DefaultTreeTableComparator** is created and added to the associated TreeTableModel. You can construct a GroupingPanel using the constructors:

```
public GroupingPanel(TreeTableModel model);
public GroupingPanel(TreeTableModel model, int axis);
public GroupingPanel(TreeTableModel model, int axis, String noGroupString);
```

You can also set and retrieve the maximum allowed number of groups with:

```
public void setMaximumGroups(int max_groups);
public int getMaximumGroups();
```

5 Treetable (new impl.)

TreeTable is the combination of a tree and a table. You can use this component in order to group several rows of a table in a single row. TreeTable extends **AdvancedJTable** and therefore inherits all of its methods and properties. See AdvancedJTable for more information. While TreeTable makes it possible to have a tree inside a table, a **TreeTableModel** is the accompanying table model that creates and maintains a tree structure. TreeTableModel contains methods for determining and changing the treetable's cell values at each tree level.

5.1 Creating

You can create a TreeTable by specifying a TreeTableModel in the constructor:

public TreeTable(TreeTableModel model);

You can also create a TreeTable by specifying a TreeTableModelAdapter:

public TreeTable(TreeTableModelAdapter adapter);

Last, using the default no-argument constructor, a TreeTable is created having a TreeTableModel that is returned by the static method:

protected static TreeTableModel createDefaultTreeTableModel();

By default, a **DefaultMutableTreeTableModel** instance is returned.

5.2 TreeTableModelAdapter

TreeTable's table model is a **TreeTableModelAdapter** which interfaces between a **TableModel** and a **TreeTableModel**. TreeTable shares the JTree that is painted on the first column of the table, with this TreeTableModelAdapter.

TreeTableModel has no knowledge of **row indexes**, but of **object nodes**. The actual row index in the table is converted by TreeTableModelAdapter to a tree node. TreeTableModelAdapter retrieves the tree node with:

public Object nodeForRow(int rowIndex);

and calls TreeTableModel's corresponding method.

The table below shows the conversion that takes place for the TableModel's methods that concern a row index.

TreeTableModelAdapter method	**TreeTableModel method**
getValueAt(**int row**, int column)	getValueAt(**Object node**, int column)
isCellEditable(**int row**, int column)	isCellEditable(**Object node**, int column)
setValueAt(Object value, **int row**, int column)	setValueAt(Object value, **Object node**, int column)

5.3 TreeTableModel

TreeTableModel is the model that is indirectly and transparently (through **TreeTableModelAdapter**) used by a **TreeTable**. TreeTableModel is an interface that extends **javax.swing.tree.TreeModel** and that has methods for querying a tree-table structure. It resembles a **TableModel** in which the methods that bear an integer row argument are replaced with an Object that defines the node at the particular tree branch. In addition, TreeTableModel has methods for determining whether a tree node is an aggregate, footer or header node. This information is supplemental to the model's function. It is basically used when deciding how to draw the cell values by table cell renderers or **Style** objects.

5.3.1 TreeTable Nodes

Most of the TreeTableModel implementations in our framework, utilize **TreeTableRow** objects as the tree's nodes. TreeTableRow extends **javax.swing.tree.DefaultMutableTreeNode**, therefore inherits all its methods and properties. A TreeTableRow provides the values for all cells in a tree node. This can be achieved in either three ways or a combination of them:

1. By using the appropriate methods of TreeTableRow.

You can assign and retrieve the value that is displayed at the i_{th} column with the methods:

```
public Object getAggregateValue(int columnIndex);
public void setAggregateValue(Object value, int columnIndex);
public void setAggregateValues(Object[] values);
```

2. By making use of DefaultMutableTreeNode's user object property.

The value at the i_{th} column can be retrieved by setting a suitable object as the user object. For example, consider the following **Customer** class:

```
public class Customer {
        String firstName;
        String lastName;
        public Customer(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
        public String getFirstName() {
                return firstName;
        }
        public String getLastName() {
                return lastName;
        }
}
```

Assuming **node** is the DefaultMutableTreeNode:

```
Customer cust = new Customer("John", "Smith");
node.setUserObject(cust);
```

The cell values are then:

```
Object userObject = node.getUserObject();
Customer cust = (Customer) userObject;
String firstName = cust.getFirstName();
String lastName = cust.getLastName();
```

3. By making use of TreeTableRow's modelIndex attribute.

The model index can correspond to an element index in a list of objects. Once the element in the list is retrieved, we can use this object to determine the cell's value:

Assuming **node** is the TreeTableRow and **objectList** the list of objects:

```
int modelIndex = node.getModelIndex();
Object o = objectList.get(modelIndex);
Customer cust = (Customer) o;
String firstName = cust.getFirstName();
String lastName = cust.getLastName();
```

5.3.2 AbstractTreeTableModel

AbstractTreeTableModel provides for an abstract treetable model that implements some common TreeTableModel methods. These are:

```
getRoot();
isAggregate(Object);
isFooter(Object);
isHeader(Object);
removeTreeModelListener(TreeModelListener);
addTreeModelListener(TreeModelListener);
```

In addition, it contains methods for creating a TreeModelEvent and passing it to its TreeModelListeners:

```
fireTreeNodesChanged(Object, Object [], int [], Object []);
fireTreeNodesInserted(Object, Object [], int [], Object []);
fireTreeNodesRemoved(Object, Object [], int [], Object []);
```

Subclasses of AbstractTreeTableModel are:

MutableTreeTableModel: allows you to dynamically add/remove tree nodes anywhere in the tree. Also, **DirectoryTreeTableModel** displays a directory treetable structure.
TreeModelMap: uses a TreeModel as the underlying tree model that contains the tree structure data.

These will be discussed later in this chapter.

5.3.3 MutableTreeTableModel

MutableTreeTableModel provides for a TreeTableModel whose nodes can be added or removed dynamically anywhere in the tree.
In order to add a node, use the following method:

public void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index);

and to remove a node:

public void removeNodeFromParent(MutableTreeNode node);

You should define the columns and their corresponding column values class upon construction time. This class assumes that javax.swing.tree.TreeNode objects will be used as the tree's nodes.

MutableTreeTableModel is an abstract class. Its subclasses should only implement the method:

public java.lang.Object getValueAt(java.lang.Object node, int column);

Next, we discuss ComparableTreeTableModel, whose tree structure is formed according to a set of rules dictated by TreeTableComparators.

5.3.3.1 ComparableTreeTableModel

ComparableTreeTableModel is a MutableTreeTableModel subclass whose tree structure is dictated by a dynamic set of rules. These rules are specified by TreeTableComparators.

ComparableTreeTableModel manages a collection of TreeTableComparators. The collection can be manipulated with the methods:

public void addRowComparator(TreeTableComparator newComparator);
public void insertRowComparator(TreeTableComparator newComparator, int index);
public TreeTableComparator removeRowComparator(int index);
public boolean removeRowComparator(TreeTableComparator comparator);
public TreeTableComparator setRowComparator(TreeTableComparator newComparator, int index);

public TreeTableComparator[] getRowComparators();
public TreeTableComparator getRowComparator(int index);

A TreeTableComparator implementation, **TreeNodeComparator**, compares two tree nodes between them based on a single column. This differs from **DefaultTreeTableComparator**, used by DynamicTreeTableModel, which compares the **row objects** of a tree node. The comparator to use for the column is retrieved with ComparableTreeTableModel's method:

public Comparator getDefaultComparator(Class columnClass);

Also, ComparableTreeTableModel installs comparators for all the common classes with the method:

protected void createDefaultComparators();

You can assign the comparator to use for a column with:

public void setDefaultComparator(Class columnClass, Comparator comparator);

ComparableTreeTableModel has methods for adding nodes to the tree, taking into account the installed comparators.

To add a new node, you can use the method:

public void addNode(DefaultMutableTreeNode node);

The methods:

public void add(List data);
public void add(Object nodeData);

can also be used for adding nodes, for which one or more leaf nodes will be created automatically with:

protected abstract DefaultMutableTreeNode createLeafNode(Object data);

Non-leaf (group) nodes will be created automatically with:

protected abstract DefaultMutableTreeNode
createNonLeafNode(DefaultMutableTreeNode child);

Finally, for re-creating the tree, when one or more comparators have changed, the

protected abstract DefaultMutableTreeNode
createNonLeafNode(DefaultMutableTreeNode child);

method needs to be implemented.

WARNING: In ComparableTreeTableModel's context, the method:

public void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index);

should never be used to add a node, unless the correct index to place the new node is known.

Next, we discuss two subclasses of MutableTreeTableModel, **DefaultMutableTreeTableModel** and **ObjectTreeTableModel**.

5.3.3.2 DefaultMutableTreeTableModel

DefaultMutableTreeTableModel is a treetable model whose nodes' cell value depends on the aggregate values of a TreeTableRow. Thus this class assumes that the nodes of the tree are TreeTableRow objects.

The value at each treetable cell is retrieved with TreeTableRow's method:

public Object getAggregateValue(int columnIndex);

5.3.3.3 ObjectTreeTableModel

ObjectTreeTableModel is a treetable model whose nodes' cell value depends on the user object attribute of a DefaultMutableTreeNode.

The user object is retrieved with DefaultMutableTreeNode's method:

public Object getUserObject();

Then, the user object is passed to the method:

public Object getObjectAt(Object userObject, int column);

, so that to return a value for the specific treetable cell.

ObjectTreeTableModel is an abstract class. Subclasses need only provide the column names and classes and implement the **getObjectAt** method described above.

5.3.4 TreeModelMap

TreeModelMap wraps around a tree model, which contains the actual tree structure data. Calls to tree model's methods are passed on to the underlying tree model. TreeModelMap also assumes that the tree's nodes are javax.swing.TreeNode objects and thus, includes methods that facilitates the creation of TreeModelEvents.

You can assign and retrieve the underlying tree model with these methods respectively:

public void setTreeModel(TreeModel newModel);
public TreeModel getTreeModel();

5.3.5 DynamicTreeTableModel

DynamicTreeTableModel is a treetable model whose tree strucutre is created dynamically every time the data stored in an underlying **ListTableModel** changes. When this occurs, DynamicTreeTableModel dynamically builds a tree, whose structure depends on the actual data and on **TreeTableComparators**,
that define the way the grouping of rows should be performed.

5.3.5.1 Creating

DynamicTreeTableModel wraps around a **ListTableModel** that provides the actual data. The ListTableModel to use is specified in the sole constructor:

public DynamicTreeTableModel(ListTableModel tableModel);

You can also set the ListTableModel instance later using the method:

public void setModel(ListTableModel newModel);

Example 1: Create a DynamicTreeTableModel and set it to a treetable.

```
//create the table models.
ListTableModel flatModel = new DefaultListTableModel();
DynamicTreeTableModel ttm = new DynamicTreeTableModel(flatModel);

//create the table
TreeTable table = new TreeTable(ttm);
```

Example 2: Create a sortable and filterable DynamicTreeTableModel and set it to a treetable.

```
//create the chain of table models.
ListTableModel flatModel = new DefaultListTableModel();
FilterTableModel ftm = new FilterTableModel(flatModel);
SortTableModel stm = new SortTableModel(ftm);
DynamicTreeTableModel ttm = new DynamicTreeTableModel(stm);

//create the table
TreeTable table = new TreeTable(ttm);
```

```
//take care of the SortTableModel's header renderer
stm.setHeader(table.getTableHeader());

//create the FilterHeaderModel
FilterHeaderModel fhm = new CustomPopupFilterHeaderModel();

//assign AdvancedJTable's header to the FilterHeaderModel
fhm.setTableHeader((FilterTableHeader) table.getTableHeader());

//attach FilterHeaderModel to the table
fhm.attachToTable(table);
```

5.3.5.2 TreeTableRows

DynamicTreeTableModel nodes consist of instances of the abstract class **TreeTableRow**, which extends **DefaultMutableTreeNode**. They are divided in:

1. **DataRows**: nodes that are associated with the actual data of the underlying ListTableModel. The association is made through the modelIndex attribute.These nodes cannot have children.
2. **AggregateRows**: nodes that are not associated with the data of the underlying ListTableModel, but that may provide information about several rows of the table. These are further classifed into:
   - **HeaderRows**: branch nodes that have children and that can be expanded. HeaderRows usually contain information about their children.
   - **FooterRows**: nodes placed at the bottom of each tree level. FooterRows usually contain information about the rows above them. They are added to the tree as long as a **Footer** is defined in the DynamicTreeTableModel.

A TreeTableRow is constructed by specifying the data row object and the index that corresponds to that object in the data list of the underlying ListTableModel:

```
public TreeTableRow(Object o, int modelIndex);
```

TreeTableRow also has methods that determine its type:

```
public boolean isAggregate();
public boolean isHeader();
public boolean isFooter();
```

Finally, there are methods for getting and setting the aggregate values:

```
public Object getAggregateValue(int rowIndex, int columnIndex);
public void setAggregateValue(Object value, int rowIndex, int columnIndex);
```

5.3.5.3 Getting to the data

DynamicTreeTableModel builds the tree structure dynamically every time the underlying data changes. To get from the "tree-view" to the original data model provided by the underlying ListTableModel, you can use the following method:

public int getDataRow(TreeTableRow node);

The above method assumes that **node** is a DataRow (no-children). In order to retrieve the indexes for HeaderRows, use:

public int[] getDataRows(TreeTableRow node);
public int[] getModelIndexesUnderRow(TreeTableRow node, boolean sorted);

Example 1: Find the objects that correspond to a treetable's row selection. (using the user object of the tree node)

```
//table is a TreeTable
int[] selectedRows = table.getSelectedRows();

TreeTableModelAdapter adapter = (TreeTableModelAdapter) table.getModel();
List treeList = ttm.getRows();
for (int i=0;i<selectedRows.length;i++) {
        TreeTableRow treeRow = (TreeTableRow)
adapter.nodeForRow(selectedRows[i]);
        Object objectRow = treeRow.getUserObject();
}
```

Example 2: Find the objects that correspond to a treetable's row selection. (using the underlying ListTableModel)

```
//table is the TreeTable model
int[] selectedRows = table.getSelectedRows();

TreeTableModelAdapter adapter = (TreeTableModelAdapter) table.getModel();
DynamicTreeTableModel ttm = (DynamicTreeTableModel)
adapter.getTreeTableModel();
ListTableModel wrappedModel = ttm.getModel();
for (int i=0;i<selectedRows.length;i++) {
        int origIndex = ttm.getDataRow(selectedRows[i]);
        Onject objectRow = wrappedModel.getRows().get(origIndex);
}
```

5.3.5.4 TreeTableComparators

TreeTableComparators are used in order to dynamically group the rows of a TreeTable

component. TreeTableComparator implements the java.util.Comparator interface, thus the method:

public int compare(Object o1, Object o2);

, should be implemented.

The supplied objects to the **compare** method above are the row data objects of the underlying ListTableModel.

DynamicTreeTableModel manages a collection of TreeTableComparators. The collection can be manipulated with the methods:

public void addRowComparator(TreeTableComparator newComparator);
public void insertRowComparator(TreeTableComparator newComparator, int index);
public TreeTableComparator removeRowComparator(int index);
public boolean removeRowComparator(TreeTableComparator comparator);
public TreeTableComparator setRowComparator(TreeTableComparator newComparator, int index);
public TreeTableComparator[] getRowComparators();
public TreeTableComparator getRowComparator(int index);

A TreeTableComparator implementation, **DefaultTreeTableComparator**, compares row data based on a single column. The comparator to use for the column is retrieved with DynamicTreeTableModel's method:

public Comparator getDefaultComparator(Class columnClass);

Also, DynamicTreeTableModel installs comparators for all the common classes with the method:

protected void createDefaultComparators();

You can assign the comparator to use for a column with:

public void setDefaultComparator(Class columnClass, Comparator comparator);

It is up to the developer to use the installed 'class' comparators with their own TreeTableComparator implementation.

Example 1: Create a TreeTableComparator that compares data based on the first column

```
//ttm is the TreeTableModel

class FirstColumnTreeTableComparator implements TreeTableComparator {
        DynamicTreeTableModel model;
        FirstColumnTreeTableComparator(DynamicTreeTableModel model) {
                this.model = model;
        }
```

```
        public int compare(Object o1, Object o2) {
                //get the cell value for column 0
                Object val1 = model.getCellValue(o1, 0);
                Object val2 = model.getCellValue(o2, 0);

                //get a comparator from the treetable model using the object's class
                Comparator c = model.getDefaultComparator(val1.getClass());

                //we can use this comparator to make the comparison
                int comparison = c.compare(val1, val2);

                //return
                return comparison;
        }
        public boolean isGroupedByColumn(int column) {
                return column == 0;
        }
};
```

Example 2: Create a TreeTableComparator that compares data based on the first and the second column

```
//ttm is the TreeTableModel

class MultipleColumnTreeTableComparator implements TreeTableComparator {
        DynamicTreeTableModel model;
        FirstColumnTreeTableComparator(DynamicTreeTableModel model) {
                this.model = model;
        }
        public int compare(Object o1, Object o2) {
                //get the cell value for column 0
                Object val1 = model.getCellValue(o1, 0);
                Object val2 = model.getCellValue(o2, 0);

                //get the cell value for column 1
                Object val3 = model.getCellValue(o1, 1);
                Object val4 = model.getCellValue(o2, 1);

                //get a comparator for the first column from the treetable model using the
object's class
                Comparator c1 = model.getDefaultComparator(val1.getClass());

                //get a comparator for the second column from the treetable model using
the object's class
                Comparator c2 = model.getDefaultComparator(val3.getClass());

                //we can use these comparators to make the comparison
                int comparison1 = c1.compare(val1, val2);
```

```
                int comparsion2 = c2.compare(val3, val4);

                //return
                return comparison1 == 0 ? comparison2 : comparison1;
        }
        public boolean isGroupedByColumn(int column) {
                return column == 0 || column == 1;
        }
};
```

Example 3: Create a TreeTableComparator that compares data based on the objects supplied. We assume that the supplied objects implement the Comparable interface.

```
//ttm is the TreeTableModel

class ComparableTreeTableComparator implements TreeTableComparator {
        public int compare(Object o1, Object o2) {
                Comparable c1 = (Comparable) o1;
                return c1.compareTo(o2);
        }
        public boolean isGroupedByColumn(int column) {
                return false;
        }
};
```

5.3.5.5 Aggregators

Aggregators calculate and return values for the aggregate rows of a TreeTable component. The value for the cell of an aggregate row is calculated with:

```
public Object getAggregateValue(AggregateRow node, int columnIndex);
```

Then, the value is assigned on the row with:

```
public Object prepare(AggregateRow row, int columnIndex);
```

, which ensures that aggregate values are not calculated repeatedly.

DynamicTreeTableModel defines methods for creating, assigning and retrieving the aggregators:

```
protected Aggregator createDefaultAggregator();
public void setDefaultAggregator(Aggregator aggregator);
public Aggregator getAggregator(int columnIndex);
public Aggregator getDefaultAggregator();
```

By default, a **DefaultCellAggregator** is created which uses the installed **TreeTableComparators** to calculate the aggregate cell values.

5.3.5.6 Footers

The **Footer** interface defines the place and number of **FooterRows** to add to a DynamicTreeTableModel via the method:

public int getFooterSize(TreeTableRow row);

Footer implementations should return the number of footers to add under the supplied TreeTableRow. An appropriate aggregator should also be used to calculate the cell values for these FooterRows.

DynamicTreeTableModel defines methods for creating, assigning and retrieving the footer:

protected Footer createDefaultFooter();
public void setFooter(Footer footer);
public Footer getFooter();

DynamicTreeTableModel will use the assigned footer in order to add footer rows to the tree structure. This is accomplished with the method:

protected void buildFooter();

Example: Create and install a custom footer and an accompanying aggregator that sums over the integers values of the cells.

```
//ttm is the TreeTableModel

//create the footer
Footer myFooter = new Footer() {
        public int getFooterSize(TreeTableRow row) {
                if (row.getLevel() == 0) return 0;
                return 1;
        }
};

//create the aggregator
Aggregator footerAggregator = new DefaultCellAggregator(ttm) {
        public Object getAggregateValue(AggregateRow node, int columnIndex) {
                if (columnIndex == 4 && node.isFooter()) {
                        TreeTableRow parent = (TreeTableRow) node.getParent();

                        int[] totalRows = model.getModelIndexesUnderRow(parent, false);
                        int sum = 0;
                        for (int i = 0; i < totalRows.length; i++) {
                                Integer iv = (Integer)
```

```
model.getModel().getValueAt(totalRows[i], 4);
                                int ival = iv.intValue();
                                sum += ival;
                        }
                        return new Integer(sum);
                }
                return super.getAggregateValue(rowIndex, columnIndex);
        }
};

//set the footer
ttm.setFooter(myFooter);

//set the aggregator
ttm.setDefaultAggregator(footerAggregator);
```

5.3.6 TreeTableModelMap

TreeTableModelMap extends **TreeModelMap** and wraps around a TreeTableModel, which contains the actual tree structure data. Calls to treetable model's methods are passed on to the underlying treetable model.

TreeTableModelMap also implements the ReorderModel, ReorderListener and CacheableTreeTableModel interfaces, to provide for caching and to be able to track the reordering of tree table rows. This functionality will be valid only if the treetable model that is passed in the constructor is also a ReorderModel and/or a CacheableTreeTableModel.

Next, we discuss two usefull subclasses, **DefaultSortTreeTableModel** and **DefaultFilterTreeTableModel**, which can sort and filter the tree nodes of a treetable model respectively.

5.3.7 Sorting

Sorting capabilities are added to a TreeTableModel by wrapping it around a **DefaultSortTreeTableModel**. This class uses an internal **SortTableModel** instance to sort the children of each group row in the underlying treetable model.

The sort tablemodel is retrieved with the method:

public SortTableModel getSortTableModel();

You can customize the sorting behaviour by manipulating this SortTableModel instance.

5.3.8 Filtering

Filtering capabilities are added to a TreeTableModel by wrapping it around a **DefaultFilterTreeTableModel**. This class uses an internal **FilterTableModel** instance to filter the children of each group row in the underlying treetable model.

The filter tablemodel is retrieved with the method:

public FilterTableModel getFilterTableModel();

You can customize the filtering behaviour by manipulating this FilterTableModel instance.

5.3.9 DirectoryTreeTableModel

DirectoryTreeTableModel is the treetable model that can hold a directory structure. DirectoryTreeTableModel displays the filename, size, type and modifed date of a file or directory. Additionally, the corresponding icon is drawn next to the filename text.

The nodes of a DirectoryTreeTableModel are either HeaderRow (the directories) or DataRow objects (the files). The files inside a directory are retrieved only when that directory is expanded. To ensure that this is done only once, DirectoryTreeTableModel takes advantage of HeaderRow's methods:

public boolean isCountEvaluated();
public void setCountEvaluated(boolean countEval);

NOTE: You can use DefaultSortTreeTableModel and DefaultFilterTreeTableModel in conjunction with DirectoryTreeTableModel to achieve a sorting and filtering effect:

```
DirectoryTreeTableModel dtm = new DirectoryTreeTableModel();
DefaultFilterTreeTableModel ftm = new DefaultFilterTreeTableModel(dtm);
DefaultSortTreeTableModel stm = new DefaultSortTreeTableModel(ftm);
```

Extra customization is needed for TreeTable to paint sorting and filtering events:

```
TreeTable table = new TreeTable();
FilterHeaderModel ft = new CustomPopupFilterHeaderModel();
ft.setFilterMode(ft.ALL_VALUES_MODE);
ft.setTableHeader((com.citra.filter.FilterTableHeader) getTableHeader());
ft.attachToTable(table, ftm.getFilterTableModel());

stm.getSortTableModel().setComparator(0, new FileRowComparator());
stm.getSortTableModel().setHeader(getTableHeader());
stm.addReorderListener(table.getTableReorder());

TreeTableModelAdapter newAdapter = new TreeTableModelAdapter(stm,
```

```
table.getTree());
table.setModel(newAdapter);
```

DirectoryTreeTable already includes the above customizations for you.

5.3.10 RemoteTreeTableModel

You can use **RemoteTreeTableModel** in order to asynchronously retrieve the data from an underlying cacheable treetable model. This is described in RemoteTreeTableModel.

5.4 Renderers

TreeTable uses separate renderers to paint the aggregate nodes of the tree. The aggregate renderer for a cell is taken with either specifying the row and column index for the cell, or the class of the cell's value:

```
public TableCellRenderer getAggregateCellRenderer(int row, int column);
public TableCellRenderer getDefaultAggregateRenderer(Class columnClass);
```

You can also assign an aggregate renderer, according to the class of the cell's value with:

```
public void setDefaultAggregateRenderer(Class columnClass, TableCellRenderer
renderer);
```

By default, TreeTable will install **DefaultTreeTableRenderer** instances for all the basic classes (String, Object, Date, Boolean and Number).

5.5 Cell Spanning

TreeTable extends AdvancedJTable and is therefore capable of cell spanning. TreeTable overrides AdvancedJTable's method:

```
protected SpanDrawer createSpanDrawer();
```

, in order to set an inner DefaultSpanModel subclass, **TreeTable.DefaultTreeSpanModel**, which spans the header rows of the treetable. Here is how DefaultTreeSpanModel is implemented:

```
public class DefaultTreeSpanModel extends DefaultSpanModel {
        private CellSpan cs;
        /**
         * Constructs a DefaultTreeSpanModel.
         */
```

```
        public DefaultTreeSpanModel() {
                cs = new CellSpan(0, 0, 0, CellSpan.ALL_COLUMNS);
        }
        public CellSpan getCellSpanAt(int row, int column) {
                TreeTableModel model = (TreeTableModel) getModel();
                if (model.isHeader(row)) {
                        cs.setSpannedRow(row);
                        return cs;
                }
                return super.getCellSpanAt(row, column);
        }
}
```

5.6 GroupingPanel

GroupingPanel is a panel through which users can dynamically control the structure of a TreeTable. The TreeTableModel must be a DynamicTreeTableModel. GroupingPanel uses a box layout in order to layout a number of comboboxes, whose items are populated with the columns of a table. By selecting a column in the combo box, the appropriate **DefaultTreeTableComparator** is created and added to the associated TreeTableModel. You can construct a GroupingPanel using the constructors:

public GroupingPanel(DynamicTreeTableModel model);
public GroupingPanel(DynamicTreeTableModel model, int axis);
public GroupingPanel(DynamicTreeTableModel model, int axis, String noGroupString);

You can also set and retrieve the maximum allowed number of groups with:

public void setMaximumGroups(int max_groups);
public int getMaximumGroups();

6 Sorting Data

You can add sorting capabilities to a table by using **SortTableModel**. SortTableModel is a **ListTableMap** that transforms the underlying 'unsorted data' into a sorted data set. The sorting is performed by comparing the data with the help of **Comparators**. The user can sort the data by clicking on a column of the table's header. SortTableModel also installs a special renderer on the table's header so that to display the sorting order and to paint the mouse clicks on the header accordingly.

6.1 Creating

SortTableModel provides the sorting by manipulating the data given to it by an underlying ListTableModel instance. See ListTableModel for more information.

There are two constructors:

SortTableModel(ListTableModel tableModel);
SortTableModel(ListTableModel tableModel, JTableHeader tableHeader);

The underlying ListTableModel can be passed as an argument in the constructor or can be specified later via SortTableModel's superclass (ListTableMap) method:

public void setModel(ListTableModel newModel);

You also need to specify the table's header for sorting to occur when the header is clicked. This can be passed as an argument in the second constructor, or can be given later with:

public void setHeader(JTableHeader newTableHeader);

Example: Create a SortTableModel

```
(1)
ListTableModel unsortedModel = new DefaultListTableModel();
SortTableModel sortedModel = new SortTableModel(unsortedModel);
JTable table = new JTable();
table.setModel(sortedModel);
sortedModel.setHeader(table.getTableHeader());

(2)
JTable table = new AdvancedJTable();
ListTableModel unsortedModel = new DefaultListTableModel();
SortTableModel sortedModel = new SortTableModel(unsortedModel,
table.getTableHeader());
table.setModel(stm);

(3)
ListTableModel unsortedModel = new DefaultListTableModel();
SortTableModel sortedModel = new SortTableModel(unsortedModel);
JTable table = new JTable(sortedModel);
sortedModel.setHeader(table.getTableHeader());
```

6.2 Comparators

SortTableModel sorts the data by comparing the cell values with each other. The comparison is

performed with special objects, called **Comparators**, which implement the java.util.Comparator interface.

Comparators compare two objects passed as an argument in the method:

public int compare(Object o1, Object o2);

, and return a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

We have implemented several comparators for all the common objects. These can be found in the com.citra.comparators package:

BooleanComparator: for Boolean values.
DateComparator: for Dates.
StringComparator: for Strings.
CaseInsensitiveStringComparator: for Strings, ignoring case differences.
GeneralComparator: for all other objects that implement the java.lang.Comparable interface.

When sorting is requested for a column, SortTableModel will try to retrieve a Comparator for that column by calling the method:

public Comparator getComparator(int column);

If this method returns null, SortTableModel will then invoke the method:

public Comparator getDefaultComparator(Class columnClass);

, by first retrieving the class of the column.

If this method returns null as well, then a NullPointerException will be thrown. To avoid this, you need to install comparators with:

public void setComparator(int column, Comparator comparator);
public void setDefaultComparator(Class columnClass, Comparator comparator);

By default, SortTableModel installs comparators for the most common classes. This is done in the method:

protected void createDefaultComparators();

Note: Comparators are also used by TreeTableModel in order to group the tabular data.

6.3 Getting to the data

After sorting, the rows of the table have been reordered. In order to retrieve this transformation, you can call the method:

public int[] getSortedIndexes();

, which returns an int array showing the relation between the original and the sorted data.

Also, since SortTableModel extends ListTableMap, the method:

public java.util.List getRows();

will return the transformed sorted list of object rows.

Furthermore, the methods:

public Object getValueAt(int row, int column);
public boolean isCellEditable(int row, int column);
public void setValueAt(Object aValue, int row, int column);
public void removeRow(int row);
public void removeRows(int[] deletedRows);

all operate on the sorted data.

The original 'unsorted' data can be retrieved by using ListTableModel's method:

public ListTableModel getModel();

6.4 Single and multi column sorting

SortTableModel is capable of using more than one columns to sort the data of a table. You can choose between single and multi-column sorting with the method:

public void setSortMode(int mode);

The int variable SortTableModel.SINGLE_SORT is for single-column, while SortTableModel.MULTI_SORT for multi-column sorting.

You can sort the rows of a table by clicking on a column header. The first time you click on a previously unsorted column, rows are sorted in ascending order. If you click again the data are sorted in descending order. A third click will result in the rows being sorted in ascending order again. In order to remove a sorting from a column, you should have the ALT key pressed while clicking. In MULTI_SORT mode, you can add sorting columns by having the CTRL key pressed while clicking on a column.

You can also use the method:

public void sort(int column, int mode);

to perform sorting programmatically. The first argument is the column to sort, while the second, the sorting mode, which can be:

ADD_SORT: clears sorting columns before adding a column to sort,
REMOVE_SORT: removes a column from sorting and
INSERT_SORT: adds a sorting column.

6.5 Define which columns can be sorted

You can define which column are sortable by calling the method:

public void setSortableColumn(int column, boolean sortable);

All columns are sortable by default.

The non-sortable columns can be retrieved with:

public int[] getNonSortableColumns();

,which returns the non-sortable columns as an int array.

You can also determine if a column can be sorted by calling:

public boolean isSortable(int column);

6.6 Controlling the visual behaviour of SortTableModel

SortTableModel installs a renderer on the table's header, so that to draw the mouse clicks, the sorting order and the sorting index. The renderer is an instance of the abstract class **SortTableRenderer**. By default, a **SortTableButtonRenderer** is employed, which uses a clickable JButton. SortTableButtonRenderer also shows the sorting order with an arrow pointing up or down accordingly, and the column sorting index as a number next to the column name.

You can use your own SortTableRenderer by overriding SortTableModel's method:

protected SortTableRenderer createDefaultSortTableRenderer();

7 Filtering Data

A **FilterTableModel** is used in order to provide filtering capabilities to a table. FilterTableModel is a **ListTableMap** that transforms the underlying 'unfiltered data' into a filtered data set. The filtering is performed according to a set of rules, implemented by **Filter** objects. In addition, a helper class, **TableFilter**, is used by FilterTableModel to carry out the whole filtering process. Finally, a **FilterModel** is used to fire **FilterModelEvents**, which are constructed via GUI classes, such as **FilterTablePanel** or **FilterHeaderModel**.

7.1 Creating

FilterTableModel filters the data presented to it by an underlying ListTableModel instance. See ListTableModel for more information.

You can construct a FilterTableModel with the constructor:

FilterTableModel(ListTableModel tableModel);

The underlying ListTableModel can be passed as an argument in the constructor or can be specified later via FilterTableModel's superclass (ListTableMap) method:

public void setModel(ListTableModel newModel);

Example: Create a FilterTableModel

(1)
```
ListTableModel unfilteredModel = new DefaultListTableModel();
FilterTableModel filteredModel = new FilterTableModel(unfilteredModel);
JTable table = new JTable();
table.setModel(filteredModel);
```

(2)
```
ListTableModel unfilteredModel = new DefaultListTableModel();
FilterTableModel filteredModel = new FilterTableModel(unfilteredModel);
JTable table = new JTable(filteredModel);
```

7.2 Filters

The filtering is performed by special objects, called **Filters**.

A filter decides whether to let an object pass through its filtering rules with the method:

public boolean accept(Object o);

, which will return true if the object matched and should not be filtered out, false otherwise.

The abstract Filter class also includes two generic methods:

public void setFilterPattern(Object filter);
public Object getFilterPattern();

, that can be used for assigning and retrieving the given filter pattern as an arbitrary java object.

The behaviour of the filter when dealing with null values is also specified with the methods:

public boolean getAcceptNull();
public void setAcceptNull(boolean acceptNull);

We have implemented filters for all the common objects:

StringFilter: for Strings.
BooleanFilter: for Boolean values.
DateFilter: for Dates.
NumericFilter: for Numbers.

You can create your own filter by extends the abstract **Filter** class.

7.3 TableFilters

TableFilters are used by FilterTableModel to filter the data.

One of TableFilter's methods:

public boolean filter(ListTableModel tableModel, Object rowData);
public boolean filter(TableModel tableModel, int row);

, can be used to perform the filtering.

TableFilters internally use filters in order to filter the data. You specify the filter that will be used in the constructor:

public TableFilter(Filter filter);
public TableFilter(Filter filter, int column);

Furthermore, you can logically subtract, add, or negate TableFilters to create more complex ones, by using **OrTableFilter, AndTableFilter** or **NotTableFilter** respectively.

7.4 Getting to the data

After filtering, some (or all) of the original rows may not be available to the table. You can retrieve this transformation by calling the method:

public int[] getFilteredIndexes();

, which will return an int array showing the relation between the original and the filtered data.

Also, since FilterTableModel extends ListTableMap, the method:

public java.util.List getRows();

will return the transformed filtered list of object rows.

Furthermore, the methods:

public Object getValueAt(int row, int column);
public boolean isCellEditable(int row, int column);
public void setValueAt(Object aValue, int row, int column);
public void removeRow(int row);
public void removeRows(int[] deletedRows);

all operate on the filtered data.

The original 'unfiltered' data can be retrieved by using ListTableModel's method:

public ListTableModel getModel();

7.5 Presenting filter options to the user

We have implemented two classes for presenting filter options to users: **FilterTablePanel** and **FilterHeaderModel**. FilterTablePanel presents filter options in a JPanel whereas FilterHeaderModel provides a filter component on each column. Both these classes are responsible for constructing a **FilterModelEvent**, which is propagated to a **FilterModel**'s list of **FilterModelListeners**. FilterTableModel, being a FilterModelListener, receives this event and filters the data accordingly.

7.5.1 VisualFilters

A **VisualFilter** is the visual representation of a **Filter**. VisualFilter uses a JPanel that contains the controls to manipulate the filter object. An instance of the panel is retrieved with:

public javax.swing.JPanel getPanel();

VisualFilter also defines methods for binding the visual controls on the panel with a TableFilter object, which is used in constructing the FilterModelEvent which is propagated to the FilterModelListeners:

public TableFilter getTableFilter();
public void setTableFilter(TableFilter tf);

We have implemented several visual filters for all the common objects:

DateVisualFilter: for Dates
StringVisualFilter: for Strings
BooleanVisualFilter: for Booleans
NumericVisualFilter: for Numbers

7.5.2 FilterTablePanel

FilterTablePanel is a panel through which FilterModelEvents are created and propagated to FilterModelListeners. FilterTablePanel contains a collection of VisualFilters, one of each is shown at a time. The appropriate VisualFilter to use for a column is taken by calling the methods:

public VisualFilter getDefaultFilter(Class columnClass);
public VisualFilter getDefaultFilter(int field);

By default, FilterTablePanel installs visual filters for objects, strings, numbers and boolean values. This is done upon initialization with the method:

protected void createDefaultFilters();

You can construct a FilterTablePanel by using one of the constructors:

FilterTablePanel(String [] fields);
FilterTablePanel(String[] fields, Class[] classes);
FilterTablePanel(TableModel);

FilterTablePanel also includes a **FilterModel** object which is responsible for sending the events to the FilterModelListeners. FilterTableModel needs to be added to that list, so that it can receive the FilterModelEvents.

Example: Create a FilterTablePanel and use it to filter the data of a table.

```
//filteredModel is the FilterTableModel

//create the panel
FilterTablePanel filterPanel = new FilterTablePanel(filteredModel);
```

```
//add filteredModel to the list of FilterModeListeners
FilterModel filterModel = filterPanel.getFilterModel();
filterModel.addFilterModelListener(filteredModel);
```

7.5.3 FilterHeaderModel

The abstract class **FilterHeaderModel** is used to provide real-time row filtering via a component that is installed on the table's header. Its subclasses need to implement methods to add this component on the column:

```
protected void removeRenderer(TableColumn aColumn);
protected void setRenderer(TableColumn aColumn);
```

PopupFilterHeaderModel, a FilterHeaderModel subclass, installs an arrow button on the column, which, when clicked, invokes a popup menu that contains available filter expression values regarding the column. Moreover, **CustomPopupFilterHeaderModel** adds a more complex custom filter.

You can use FilterHeaderModel in any JTable with either of the methods:

```
public void attachToTable(JTable table);
public void attachToTable(JTable table, FilterListModel flm);
```

FilterHeaderModel contains a specialized JTableHeader subclass, **FilterTableHeader**, which replaces the table's header upon calling the **attachToTable** method above. You can create, retrieve and assign the header with:

```
protected FilterTableHeader createTableHeader();
public FilterTableHeader getTableHeader();
public void setTableHeader(FilterTableHeader header);
```

Note: AdvancedJTable already provides a FilterTableHeader subclass, which must be assigned to FilterHeaderModel prior to calling the **attachToTable** method. (see example 2 below)

Example 1: Install a FilterHeaderModel on a JTable

```
//first create the FilterTableModel and JTable
ListTableModel unfilteredModel = new DefaultListTableModel();
FilterTableModel filteredModel = new FilterTableModel(unfilteredModel);
JTable table = new JTable(filteredModel);

//create the FilterHeaderModel
FilterHeaderModel fhm = new CustomPopupFilterHeaderModel();

//attach FilterHeaderModel to the table
```

```
fhm.attachToTable(table);
```

Example 2: Install a FilterHeaderModel on an AdvancedJTable

```
//first create the FilterTableModel and AdvancedJTable
ListTableModel unfilteredModel = new DefaultListTableModel();
FilterTableModel filteredModel = new FilterTableModel(unfilteredModel);
JTable table = new AdvancedJTable(filteredModel);

//create the FilterHeaderModel
FilterHeaderModel fhm = new CustomPopupFilterHeaderModel();

//assign AdvancedJTable's header to the FilterHeaderModel
fhm.setTableHeader((FilterTableHeader) table.getTableHeader());

//attach FilterHeaderModel to the table
fhm.attachToTable(table);
```

8 Caching

A TableModel or TreeTableModel can be made cacheable by implementing the **CacheableTableModel** or **CacheableTreeTableModel** interface respectively. A **Cache** object can be used in conjuction with these classes, as a place to store the cached values.

8.1 CacheableTableModel

CacheableTableModel contains methods for determining whether the row count or certain cell values have been retrieved (cached). These are:

```
public boolean isCountCached();
public boolean isValueCached(int row, int column);
```

In addition, the method:

```
public boolean isRangedModel();
```

, determines whether or not the model can fetch data in ranges.

Finally, the method:

```
public java.util.List getUncachedRows(int from, int to);
```

, retrieves the cell values from the table model by specifying a row interval. In our framework, this method is used by a Cache (TableCache) object.

8.2 CacheableTreeTableModel

CacheableTreeTableModel contains methods for determining whether the children count or the column values of a tree node have been retrieved (cached). These are:

public boolean isCountCached(Object node);
public boolean isValueCached(Object node, int column);

In addition, the method:

public boolean isRangedModel();

, determines whether or not the model can fetch data in ranges.

Finally, the methods:

public java.util.List getUncachedGroups(Object parent, int from, int to);
public java.util.List getUncachedChildren(Object parent, int from, int to);
public int getUncachedGroupCount(Object node);

, retrieve the cell values and children count from the treetable model by specifying a row interval. In our framework, these methods are used by a Cache (TreeTableCache) object.

8.3 Cache

Cache represents a cache store. You can use default implementations for tables and treetables by creating a **DefaultTableCache** and **DefaultTreeTableCache** respectively.

The cache implementations have some common behaviour: you can define how the data is retrieved from the uncached model, by specifying the **maximum cache size** and the number of values to retrieve when fetching (**chunk size**).

8.4 CachedListTableModel

CachedListTableModel provides a caching behaviour for a ListTableModel. This class uses a DefaultTableCache in order to store the table's rows.

CachedListTableModel wraps around a ListTableModel, passed in the constructor:

ListTableModel uncachedModel = new DefaultListTableModel();
CachedListTableModel cacheModel = new CachedListTableModel(uncachedModel);

You can also specify the TableCache and/or the chunk size and the maximum cache size at construction time.

public CachedListTableModel(ListTableModel model, DefaultTableCache cache);
public CachedListTableModel(ListTableModel model, int chunkSize, int maximumCacheSize);

8.5 CachedTableModel

CachedTableModel provides a caching behaviour for a TableModel. This class uses a DefaultTableCache in order to store the table's rows.

CachedTableModel wraps around a TableModel, passed in the constructor:

ListTableModel uncachedModel = new DefaultListTableModel();
CachedTableModel cacheModel = new CachedTableModel(uncachedModel);

You can also specify the TableCache and/or the chunk size and the maximum cache size at construction time.

public CachedTableModel(ListTableModel model, DefaultTableCache cache);
public CachedTableModel(ListTableModel model, int chunkSize, int maximumCacheSize);

9 GroupTableHeader

GroupTableHeader provides for a TableHeader that is able to group table columns together.

9.1 GroupTableColumn

GroupTableColumn is a **TableColumn** subclass that contains children TableColumns. In addition, it holds a reference to its parent GroupTableColumn. The group table columns have a tree-like structure, with a GroupTableColumn as the root column. This tree is effectively created and managed by GroupTableHeader.

You can create a GroupTableColumn by supplying the header value to display, the default width and the renderer, editor that is installed.

e.g.

GroupTableColumn gtc = new GroupTableColumn("Personal Details");

You can add/remove table columns or group table columns as children with:

public void addColumn(TableColumn aColumn);
public void removeColumn(int columnIndex);
public void removeColumn(TableColumn aColumn);

Columns can be added to the group table column after they have been created by the table or the table column model.

Additionally, you can control the children columns visibilty with:

public boolean getShowChildren();
public void setShowChildren(boolean showChildren);

Example: Create a GroupTableColumn

```
GroupTableColumn gtc = new GroupTableColumn("Name");
gtc.addColumn(table.getColumnModel().getColumn(0)); //refers to the normal column at index 0
gtc.addColumn(table.getColumnModel().getColumn(1)); //refers to the normal column at index 1
```

9.2 GroupTableColumnModel

GroupTableColumnModel defines the requirements for a table column model object suitable for use with a **GroupTableHeader**. This interface defines methods for adding/removing **GroupTableColumnModelListeners** that are notified each time a group column is added or removed from the model.

The root group column is retrieved with:

public GroupTableColumn getRootGroupColumn();

You can use this root group column to add group columns to the model:

```
GroupTableColumn root = model.getRootGroupColumn();

GroupTableColumn gtc = new GroupTableColumn("Name");
gtc.addColumn(table.getColumnModel().getColumn(0));
gtc.addColumn(table.getColumnModel().getColumn(1));

root.addColumn(gtc);
```

9.3 GroupTableColumnModelListener

GroupTableColumnModelListener defines the interface for an object that listens to changes in a **GroupTableColumnModel**.

DefaultGroupTableColumnModel will create and send **GroupTableColumnModelEvents** to its listeners upon the removal or addition of group table columns.

9.4 Usage

You create and install a GroupTableHeader as you normally would with a JTableHeader:

```
GroupTableHeader gth = new GroupTableHeader();
JTable table = new JTable();
table.setTableHeader(gth);
```

Then you can use GroupTableHeader's methods:

```
public void addGroupColumn(GroupTableColumn aColumn);
public void removeGroupColumn(GroupTableColumn aColumn);
```

, in order to add/remove column groups.

AdvancedJTable creates its own GroupTableHeader subclass, **AdvancedJTable.InnerTableHeader**. Therefore, you need not and should not set a GroupTableHeader if you are using AdvancedJTable.

Example 1: Usage of GroupTableHeader with a JTable

```
GroupTableHeader groupHeader = new GroupTableHeader();

GroupTableColumn nameColumn = new GroupTableColumn("Name");
nameColumn.addColumn(table.getColumnModel().getColumn(0));
nameColumn.addColumn(table.getColumnModel().getColumn(1));

groupHeader.addGroupColumn(nameColumn);

table.setTableHeader(groupHeader);
```

Example 2: Usage of GroupTableHeader with AdvancedJTable

```
GroupTableHeader groupHeader = (GroupTableHeader) table.getTableHeader();

GroupTableColumn nameColumn = new GroupTableColumn("Name");
nameColumn.addColumn(table.getColumnModel().getColumn(0));
```

```
nameColumn.addColumn(table.getColumnModel().getColumn(1));

groupHeader.addGroupColumn(nameColumn);
```

10 Asynchronous Transfers (RemoteModels)

By using classes from the **com.citra.table.remote** package, the table retrieves data from its table model asynchronously. In this way, the paint thread is not blocked, and the user interface is not 'frozen'. This is extremely useful for situations where data is read from a database or from the network.

10.1 RemoteTableModel

RemoteTableModel is used to asynchronously retrieve the data from an underlying cacheable table model. Its default implementation, **DefaultRemoteTableModel** wraps around a **CacheableTableModel** and uses an internal thread in order to asynchronously retrieve the data from the underlying (uncached) model.

Example: Use a DefaultRemoteTableModel in order to asynchronously retrieve the data of an uncached DefaultTableModel:

```
DefaultTableModel dtm = new DefaultTableModel();
CachedTableModel ctm = new CachedTableModel(dtm);
DefaultRemoteTableModel rtm = new DefaultRemoteTableModel(ctm);

JTable table = new JTable();
table.setModel(rtm);
```

10.2 RemoteTreeTableModel

RemoteTreeTableModel is used to asynchronously retrieve the data from an underlying cacheable treetable model. Its default implementation, **DefaultRemoteTreeTableModel** wraps around a **CacheableTreeTableModel** and uses an internal thread in order to asynchronously retrieve the data from the underlying model.

Example: Use a DefaultRemoteTreeTableModel in order to asynchronously retrieve the data of a DirectoryTreeTableModel:

```
DirectoryTreeTableModel dtm = new DirectoryTreeTableModel();
DefaultRemoteTreeTableModel rtm = new DefaultRemoteTreeTableModel(dtm);

TreeTable table = new TreeTable();
```

```
TreeTableModelAdapter newAdapter = new TreeTableModelAdapter(rtm,
table.getTree());
table.setModel(newAdapter);
```

10.3 RemoteTableListener

RemoteTableListener is a listener that is notified each time the **RemoteTableModel** will start or stop querying the underlying table model.

Remote(Tree)TableModel will create and send **RemoteTableEvents** to its listeners upon the begining or end of an asynchronous query to the underlying model.

10.4 StatusPanel

StatusPanel is a JPanel that shows the current status of a RemoteTableModel or RemoteTreeTableModel. StatusPanel contains a label and an indicator that update themselves according to the RemoteTableEvent received.

10.5 Pending Value

PendingValue is an interface that classes implementing it are considered to represent objects that have not yet been evaluated by RemoteTableModel or RemoteTreeTableModel. DefaultRemoteTableModel and DefaultRemoteTreeTableModel use a DefaultPendingValue that its toString() method returns a certain string.

10.6 Style

You can add **RemoteStyle** to AdvancedJTable's style model so that the pending cells will be painted with a specified background color. This style is applied only if the given cell value implements the PendingValue interface.

You can assign and retrieve the pending background color with:

```
public void setPendingBackgroundColor(java.awt.Color newPendingBackgroundColor);
public java.awt.Color getPendingBackgroundColor();
```

11 Locked Rows/Columns

You can make the rows and columns at the edges of a table locked in place (non-scrollable). For

this behaviour, the table must be enclosed by an **AdvancedJScrollPane**. You can then manipulate AdvancedJScrollPane's locked table model attribute in order to assign the number of locked rows/columns:

```
AdvancedJScrollPane scroller = new AdvancedJScrollPane();
LockedTableModel lockedModel = scroller.getLockedModel();
lockedModel.setLockedColumns(2, LockedTableModel.LEFT_DIRECTION);
```

11.1 LockedTableModel

LockedTableModel is the model that holds the number of locked rows/columns at the 4 edges of the table.

You can set the locked rows/column with:

```
public void setLockedColumns(int columns, int direction);
public void setLockedRows(int rows, int direction);
```

Also, retrieve the number of locked rows/columns:

```
public int getLockedColumns(int direction);
public int getLockedRows(int direction);
```

There are four defined directions:

LEFT_DIRECTION, RIGHT_DIRECTION, TOP_DIRECTION and BOTTOM_DIRECTION.

11.2 LockedTableModelListener

LockedTableModelListener is the listener that is notified upon changes to a LockedTableModel.

DefaultLockedTableModel will create and send **LockedTableModelEvents** to its listeners upon the assignment of locked rows/columns.

11.3 Usage

You can assign the number of locked rows/columns by manipulating AdvancedJScrollPane's locked model as follows:

```
AdvancedJScrollPane scroller = new AdvancedJScrollPane();
LockedTableModel lockedModel = scroller.getLockedModel();
```

```
lockedModel.setLockedColumns(2, LockedTableModel.RIGHT_DIRECTION);
lockedModel.setLockedRows(1, LockedTableModel.TOP_DIRECTION);

JTable table = new JTable();
scroller.setViewPortView(table);
```

AdvancedJScrollPane creates and manages a LockedTableModel instance. The following AdvancedJScrollPane's methods are used for creating, getting and setting the locked table model:

```
protected LockedTableModel createDefaultLockedModel();
public LockedTableModel getLockedModel();
public void setLockedModel(LockedTableModel lockedModel);
```

NOTE: Since 3.3.5.2, AdvancedJTable no longer uses a LockedTableModel instance. The methods that concern LockedTableModel have been moved to AdvancedJScrollPane.

12 Cell Spanning

You can span the cells of a JTable by using the classes in the **com.citra.table.span** package. More specifically, **SpanDrawer** is used to draw the spanned cells according to the data fed to it by a **SpanModel**.

12.1 SpanDrawer

SpanDrawer is responsible for drawing the spanned cells of a JTable, by querying a SpanModel in order to find out which cells to span.

SpanDrawer has three constructors:

```
SpanDrawer();
SpanDrawer(JTable table);
SpanDrawer(JTable table, SpanModel spanModel);
```

You can use the third constructor to specify the associated table and the span model to use. Nevertheless, you can assign these variables later by calling:

```
public void setTable(JTable newTable);
public void setSpanModel(SpanModel newSpanModel);
```

If you do not specify a span model in the constructor, a **DefaultSpanModel** is created, through which spanned cells can be dynamically added/removed.

You can get an instance of the span model with:

public SpanModel getSpanModel();

Cell spanning is not enabled by default when you first create a SpanDrawer instance. In order to enable cell spanning, use:

public void setUseSpan(boolean useSpan);

You can also determine if cell spanning is enabled with:

public boolean getUseSpan();

AdvancedJTable creates an internal SpanDrawer object upon initialization, which is used to draw the spanned cells. In AdvancedJTable, there are methods for creating, setting and getting the span drawer to use:

protected SpanDrawer createSpanDrawer();
public void setSpanDrawer(SpanDrawer drawer);
public SpanDrawer getSpanDrawer();

Example: Manipulate AdvancedJTable's SpanDrawer object.

```
AdvancedJTable table = new AdvancedJTable();
//initialize the table with some data here.
...
...
SpanDrawer drawer = table.getSpanDrawer();
DefaultSpanModel dsm = (DefaultSpanModel) drawer.getSpanModel();
CellSpan cellSpan = new CellSpan(0, 0, 0, CellSpan.ALL_COLUMNS);
dsm.addCellSpan(cellSpan);
```

You can also use SpanDrawer in your own JTable subclass by using appropriate code. To find out more see the Appendix.

12.2 SpanModel

SpanDrawer queries a SpanModel in order to discover which cells of the table are spanned so that it can draw them appropriately. SpanModel's method:

public CellSpan getCellSpanAt(int row, int column);

is used to return a CellSpan object that specifies which cells are spanned.

CellSpan's structure is simple. A CellSpan object has four attributes:

spannedRow, spannedColumn: in a series of cell spans, these two integers define the top-left cell where the span begins.

rowSpan, columnSpan: these two integers define the number of rows and columns that the cell spans respectively.

Note: In a series of spanned cells, the getCellSpanAt method should return the same CellSpan object for all the affected cells.

If no SpanModel is specified in SpanDrawer's constructor, a **DefaultSpanModel** is created. DefaultSpanModel has the ability to dynamically add/remove spanned cells through the methods:

```
public void addCellSpan(CellSpan cellSpan);
public void removeAllCellSpans();
public void removeCellSpan(CellSpan cellSpan);
public void removeCellSpan(int row, int column);
```

You can create your own SpanModel by either implementing the SpanModel interface, or by extending the AbstractSpanModel class.

Example: Create a SpanModel that spans cells every three rows:

```
SpanModel spanModel = new AbstractSpanModel() {
        public CellSpan getCellSpanAt(int row, int column) {
                if (row % 3 == 0) {
                        return new CellSpan(row, column, 0, 4);
                }
        }
};
SpanDrawer drawer = table.getSpanDrawer();
drawer.setSpanModel(spanModel);
```

12.3 SpanModelEvent and SpanModelListener

DefaultSpanModel fires a SpanModelEvent every time a cell span is added or removed. You can listen for SpanModelEvents by creating a class that implements SpanModelListener and adding it to the SpanModel's SpanModelListeners list. For example, SpanDrawer contains an internal SpanModelListener that repaints the table's area that was affected by a cell span being added/removed.

Example: Create a SpanModelListener.

```
SpanModelListenerer mySpanModelListener = new SpanModelListener() {
        public void spanChanged(SpanModelEvent e) {
                int type = e.getType();
                CellSpan cs = e.getCellSpan();
                if (type = e.INSERT) {
                        …
                }
```

```
                if (type = e. DELETE) {
                        …
                }
                if (type = e. UPDATE) {
                        …
                }
        }
};
SpanDrawer drawer = table.getSpanDrawer();
SpanModel spanModel = drawer.getSpanModel();
spanModel.addSpanModelListener(mySpanModelListener);
```

13 Styles

Styles can beautify your table by modifying a table's renderer component, just before the component is shown on the table. While the renderer of the cell, a **TableCellRenderer** object, transforms a value to a visual component, a style performs additional actions to the component, such as changing its background color or its text font.

13.1 Creating

You can create your own Style by implementing the Style interface and implementing the method:

public void apply(java.awt.Component c, javax.swing.JTable table, int row, int column);

Example: Create a style that paints alternate cells with a red background color.

```
Style myStyle = new Style() {
        public void apply(java.awt.Component c, javax.swing.JTable table, int row, int
column) {
                if (row % 1 == 0 && column % 1 == 0) {
                        c.setBackGround(Color.red);
                }
        }
};
```

13.2 DefaultStyle

DefaultStyle is a style that paints alternate rows of a table with different colors.

You can assign the even/odd color with:

public void setEvenColor(Color evenColor);
public void setOddColor(Color oddColor);

You can also retrieve the assigned even/odd color with:

public Color getEvenColor();
public Color getOddColor();

AdvancedJTable creates a DefaultStyle upon initialization. You can specify the odd and even row colors by using AdvancedJTable's methods:

public void setEvenColor(Color evenColor);
public void setOddColor(Color oddColor);

13.3 StyleModel

A collection of Styles is kept in a StyleModel.
StyleModel iterates through its collection of styles to apply them to the Component being drawn using the method:

public void applyStyles(java.awt.Component c, javax.swing.JTable table, int row, int column);

StyleModel also includes methods for managing its internal styles list:

public void addStyle(Style newStyle);
public void clearStyles();
public Style getStyle(int index);
public Style[] getStyles();
public void insertStyle(Style newStyle, int index);
public void removeStyle(Style style);

DefaultStyleModel is the default style model implementation. An instance of it is created and used by AdavancedJTable.
AdavancedJTable defines methods for creating, getting and setting the StyleModel:

protected StyleModel createDefaultStyleModel();
public StyleModel getStyleModel();
public void setStyleModel(StyleModel styleModel);

If you want to use a StyleModel in your custom JTable, you should override the method:

public Component prepareRenderer(TableCellRenderer renderer, int row, int column);

to look like:

```
public Component prepareRenderer(TableCellRenderer renderer, int row, int column) {
        Component c = super.prepareRenderer(renderer, row, column);
        styleModel.applyStyles(c, this, row, column);
        return c;
}
```

14 JTableRowHeader

JTableRowHeader represents a vertical table header, that can be used as the row header of a JTable. This class extends JTableHeader, therefore it inherits all of JTableHeader's methods and properties. JTableRowHeader uses a TableColumnModel to manage its cells. Each cell in the header has the same column width, whereas the cell's row height is determined by the row height of the cell in the associative table.

14.1 Creating

JTableRowHeader has two constructors:

```
JTableRowHeader(TableColumnModel columnModel);
JTableRowHeader(TableColumnModel columnModel, int columnWidth);
```

You can use a **DefaultTableColumnModel** as the column model. If you do not use the second constructor to specify the column width, a default width of 25 pixels is used.

An instance of this class is created and employed by AdvancedJTable. The following AdvancedJTable's methods are used for creating, getting and setting the row header:

```
protected JTableRowHeader createDefaultTableRowHeader();
public JTableRowHeader getTableRowHeader();
public void setTableRowHeader(JTableRowHeader rowHeader);
```

You can use JTableRowHeader in any JTable by setting it as the row header view of the enclosing JScrollPane:

```
DefaultTableColumnModel columnModel = new DefaultTableColumnModel();
JTableRowHeader rowHeader = new JTableRowHeader(columnModel);

JTable table = new JTable();
rowHeader.setTable(table);

JScrollPane scroller = new JScrollPane();
scroller.setViewportView(table);
scroller.setRowHeaderView(rowHeader);
```

14.2 Controlling the visual appearance

Just as in JTableHeader, the cells are rendered by a **TableCellRenderer** object. The following JTableRowHeader's methods are used for creating, getting and setting the cell renderer:

protected TableCellRenderer createDefaultRowRenderer();
public TableCellRenderer getDefaultRowRenderer();
public void setDefaultRowRenderer(TableCellRenderer defaultRenderer);

By default, JTableRowHeader uses a **DefaultRowHeaderRenderer** to render its cells. The cells in a DefaultRowHeaderRenderer are rendered as a JButton, which text is set to the string value of the cell.

The value of each cell is determined by the method:

protected Object getColumnHeaderValue(int rowIndex);

By default, this method returns the current row number as a string.

14.3 Setting the column width

You can use the constructor:

JTableRowHeader(TableColumnModel columnModel, int columnWidth);

, in order to specify the column width.

Alternatively, you can use the method:

public void setColumnWidth(int columnWidth);

Finally, you can retrieve the current column width with:

public int getColumnWidth();

14.4 Controlling the row header's visibility

AdvancedJTable defines two methods for retrieving and assigning the header row's visibility:

public boolean getShowRowHeader();
public void setShowRowHeader(boolean show);

You can show or hide the row header in your own JTable class by manipulating the enclosing JScrollPane:

```
JScrollPane scroller = new JScrollPane();

//show the row header
scroller.setRowHeaderView(rowHeader);

//hide the row header
scroller.setRowHeaderView(null);
```

15 TreeTableHeader

TreeTableHeader is a specialized JTableHeader capable of keeping a hierarchical (tree) structure of table columns. The columns that have children contain an expand/collapse icon on the left, which can be used to expand/collapse the column respectively. The column tree structure is defined in a **TreeTableColumnModel**, which defines methods for retrieving, adding and removing table columns. Finally, TreeTableHeader's column model is a **TreeTableColumnModelAdapter** that provides an interface from the visible columns in the header to the actual column data.

15.1 TreeTableColumnModel

TreeTableColumnModel defines the structure of a hierarchical column model. TreeTableColumnModel extends **javax.swing.tree.TreeModel** and defines three extra methods for retrieving, adding and removing table columns. These are:

public TableColumn getColumn(Object node): Returns the table column at the specified node.
public Object insertColumnInto(TableColumn aColumn, Object columnNode, Object parentNode, int index): Inserts a table column at the node specified by **columnNode**, whose parent is **parentNode** and **index** is the child location in the parent node. The **columnNode** parameter is optional, by setting this to null, TreeTableColumnModel should create a new node for the supplied column. The object returned is the newly created node, or the columnNode object passed as a parameter to the method.
public void removeColumnFrom(Object node): Removes the node, and thereby the column, from the model.

Note that you can install TreeModelListeners to be notified when a column was added, removed, or changed:

```
TreeTableColumnModel.addTreeModelListener(new javax.swing.event.
TreeModelListener() {
        public void treeNodesChanged(javax.swing.event.TreeModelEvent e) {
```

```
        }
        public void treeNodesInserted(javax.swing.event.TreeModelEvent e) {
                Object[] children = e.getChildren();
                if (children == null) return;
                for (int i = 0; i < children.length; i++) {
                        TableColumn tc = treeColumnModel.getColumn(children[i]);
                        //column tc was added
                }
        }
        public void treeNodesRemoved(javax.swing.event.TreeModelEvent e) {

        }
        public void treeStructureChanged(javax.swing.event.TreeModelEvent e) {

        }
});
```

15.2 DefaultTreeTableColumnModel

DefaultTreeTableColumnModel provides a default implementation for a TreeTableColumnModel. For not rewriting the code associated to dealing with tree models, we made it extend ObjectTreeTableModel, although the extra 'column' dimension introduced by TreeTableModel is not used as it is not needed. DefaultTreeTableColumnModel uses TableColumns as the user objects of its (DefaultMutableTreeNode) nodes. Additionally, the TreeTableColumnModel methods are actually calls to its **MutableTreeTableModel** superclass.

Note that, when inserting columns, TableColumn's modelIndex attribute must be associated to the table's data model. For example, if you use the no-argument TableColumn constructor, all created columns will bear a model index of 0, thereby referring to the first column in the data model.

Example: Use DefaultTreeTableColumnModel to create a three level tree header

```
//group
DefaultTreeTableColumnModel model = new DefaultTreeTableColumnModel();

//first level
TableColumn continents = new TableColumn(0);
continents.setHeaderValue("Continents");
Object continentsNode = model.insertColumnInto(continents, null, model.getRoot(), 0);

//second level - europe
TableColumn europe = new TableColumn(1);
europe.setHeaderValue("Europe");
Object europeNode = model.insertColumnInto(europe, null, continentsNode, 0);
```

```
//second level - america
TableColumn america = new TableColumn(2);
america.setHeaderValue("America");
Object americaNode = model.insertColumnInto(america, null, continentsNode, 1);
//second level - asia
TableColumn asia = new TableColumn(3);
asia.setHeaderValue("Asia");
Object asiaNode = model.insertColumnInto(asia, null, continentsNode, 2);

//third level - europe/germany
TableColumn germany = new TableColumn(4);
germany.setHeaderValue("Germany");
Object germanyNode = model.insertColumnInto(germany, null, europeNode, 0);
//third level - europe/uk
TableColumn uk = new TableColumn(5);
uk.setHeaderValue("UK");
Object ukNode = model.insertColumnInto(uk, null, europeNode, 1);
//third level - europe/italy
TableColumn italy = new TableColumn(6);
italy.setHeaderValue("Italy");
Object italyNode = model.insertColumnInto(italy, null, europeNode, 2);

//third level - america/USA
TableColumn usa = new TableColumn(7);
usa.setHeaderValue("USA");
Object usaNode = model.insertColumnInto(usa, null, americaNode, 0);
//third level - america/colombia
TableColumn colombia = new TableColumn(8);
colombia.setHeaderValue("Colombia");
Object colombiaNode = model.insertColumnInto(colombia, null, americaNode, 1);
//third level - america/cuba
TableColumn cuba = new TableColumn(9);
cuba.setHeaderValue("Cuba");
Object cubaNode = model.insertColumnInto(cuba, null, americaNode, 2);
```

Note that the int argument passed to the TableColumn constructor is the column index that maps to the column in the table's data model (the TableModel).

15.3 TreeTableColumnModelAdapter

TreeTableColumnModelAdapter is the column model used by TreeTableHeader. TreeTableColumnModelAdapter maintains an instance to the TreeTableColumnModel containing the column data, while it exposes, with the help of a JTree, the expanded columns to the table. The collapsed columns remain hidden and are made available to the table upon tree expansion events.

TreeTableColumnModelAdapter has two constructors:

public TreeTableColumnModelAdapter(): Default, no-argument, a TreeTableColumnModel is created with **protected TreeTableColumnModel createDefaultTreeColumnModel().**
public TreeTableColumnModelAdapter(TreeTableColumnModel treeTableModel): assigns the TreeTableColumnModel at construction time.

The TreeTableColumnModel can be assigned any time using the method:

public void setTreeTableColumnModel(TreeTableColumnModel model)

Note that TreeTableHeader will create a TreeTableColumnModelAdapter upon initialization.

TreeTableColumnModelAdapter also has a JTree which is used to find out which column nodes are expanded or collapsed. You can retrieve the JTree variable with the method:

public JTree getTree()

The JTree can then be used to expand or collapse rows or tree paths programatically:

```
JTree tree = TreeTableColumnModelAdapter.getTree();
tree.expandRow(1);
```

You can also install TreeExpansionListeners to be notified after a column has been expanded:

```
tree.addTreeExpansionListener(new TreeExpansionListener() {
        public void treeExpanded(TreeExpansionEvent event) {
                TreePath path = event.getPath();
                Object node = path.getLastPathComponent();
                //supposing model is the TreeTableColumnModel
                TableColumn column = model.getColumn(node); //column has been
expanded
        }
        public void treeCollapsed(TreeExpansionEvent event) {

        }
});
```

Also, to find the node at a specific row of the tree, you can use the method:

public Object nodeForRow(int rowIndex)

15.4 Usage

To install a TreeTableHeader on a table, the TreeTableColumnModel that will contain the

hierarchical column structure must first be created. Since, by assigning the tree header to the table, will make the table assign its 'flat' column model to the header, the TreeTableHeader's column model must be assigned to the table **prior** to assigning the header to the table. Otherwise, a ClassCastException will be thrown.

Example:

```
DefaultTreeTableColumnModel model = new DefaultTreeTableColumnModel();

//fill the model with some values as in this example

//create the TreeTableHeader
TreeTableHeader treeHeader = new TreeTableHeader(model);

//create the table
AdvancedJTable table = new AdvancedJTable();

//assign the 'tree' column model of the tree header - THIS IS IMPORTANT
table.setColumnModel(treeHeader.getColumnModel());

//assign the tree header to the table
table.setTableHeader(treeHeader);
```

16 CheckBoxTree

CheckBoxTree is a JTree subclass that carries a checkbox at every node in the tree. The checkbox can be selected/unselected both visually and programmatically. The checkbox's selections are stored and managed by a **CheckBoxTreeSelectionModel** whose default implementation is **DefaultCheckBoxTreeSelectionModel**.

16.1 CheckBoxTreeSelectionModel

CheckBoxTreeSelectionModel is the checkbox's selection model. It can be used to find out what tree nodes are selected and to select/deselect one or more nodes. CheckBoxTreeSelectionModel extends TreeSelectionModel, which defines the selection model for a JTree. The difference here is that while TreeSelectionModel treats paths as tree rows being selected, CheckBoxTreeSelectionModel's selections paths refer to the nodes whose checkbox is selected.

You can use TreeSelectionModel's methods to manage the checkboxes selection, just as you do for the tree rows selections.

For example, the method to retrieve all selected paths is:

public TreePath[] getSelectionPaths();

To find out if a specific path is selected:

public boolean isPathSelected(TreePath path);

To add a checkbox selection:

public void addSelectionPath(TreePath path);

To remove a checkbox selection:

public void removeSelectionPath(TreePath path);

To remove all selected paths:

public void clearSelection();

Install a listener that is notified every time a checkbox selection is changed:

public void addTreeSelectionListener(TreeSelectionListener x);

16.2 Usage

You can create a CheckBoxTree just as you create a normal JTree. For example:

CheckBoxTree tree = new CheckBoxTree(new Object[]{"one", "two", "three"});

or

CheckBoxTree tree = new CheckBoxTree(new DefaultTreeModel());

The CheckBoxTreeSelectionModel is retrieved and assigned respectively with:

public CheckBoxTreeSelectionModel getCheckBoxSelectionModel();
public void setCheckBoxSelectionModel(CheckBoxTreeSelectionModel checkSelectionModel);

You can also change the gap that exists between the checkbox and the node's value with the method:

public void setCheckBoxGap(int checkBoxGap);

CheckBoxTree can also be assigned to a TreeTable:

```
CheckBoxTree tree = new CheckBoxTree();
TreeTable table = new TreeTable();
table.getTreeTableModelAdapter().setTree(tree);
```

17 TreeFilterHeaderModel

TreeFilterHeaderModel provides filtering to a TreeTableModel through a component that is installed on the column header of a TreeTable. TreeFilterHeaderModel is an abstract class whose default implementation is **PopupTreeFilterHeaderModel**. The available filter expressions for a column are handled by a ColumnFilterMapper, which has three implementations:
1. **NodeFilterMapper** displays every node of the TreeTableModel
2. **LevelFilterMapper** displays unique values at each node level
3. **DefaultColumnFilterMapper** combines NodeFilterMapper and LevelFilterMapper so that you can define the association of each column with one of the two.

By default, PopupTreeFilterHeaderModel uses a DefaultColumnFilterMapper.

17.1 ColumnFilterMapper

ColumnFilterMapper is used to present users with available filter expressions and to create, apply and remove filters for a FilterTreeTableModel.
The method that is called before filter expressions are made available to the user is:

public void filterVisible(int modelIndex, CheckBoxTree tree, TreeTableModel treeTableModel, TreeTable table);

The above method will prepare the TreeTable accordingly, with values from the given treeTableModel parameter.

Then, if the filter action is commited, the following method is called:

public void commitFilters(FilterTreeTableModel filterTreeTableModel, Filter filter, int modelIndex);

Otherwise, if the filter action is cancelled, the method below is called:

public void filterCancelled();

The stored filters can be applied any time for any FilterTreeTableModel with the method:

public void applyFilters(FilterTreeTableModel filterTreeTableModel, int[] excludedModelIndexes);

They can also be removed with:

public boolean removeFilters(int[] modelIndexes);

Additionally, to determine whether a given column has a filter:

public boolean hasFilter(int modelIndex);

17.2 Usage

TreeFilterHeaderModel bears some similarities to FilterHeaderModel. In order to install it to a TreeTable, you should call:

public void attachToTable(TreeTable treetable) OR
public void attachToTable(TreeTable treetable, FilterTreeTableModel ftm);

You can add and remove 'filterable' columns respectively with:

public void setFilterControlInColumn(TableColumn aColumn);
public void removeFilterControlFromColumn(TableColumn aColumn);

By default, all columns are 'filterable'. This can be controlled with:

public void setAutoCreateAllFilters(boolean newAutoCreateAllFilters);
public boolean getAutoCreateAllFilters();

The Filter that will be used to filter the values of a column is retrieved with the method:

public Filter getFilter(int viewIndex);

The method above will call:

public Filter getDefaultFilter(Class columnClass);

You can set the default filters for a Class with:

public void setDefaultFilter(Class columnClass, Filter filter);

By default, Filters for all the common objects (Strings, Dates, Booleans etc) are installed at construction time with:

protected void createDefaultFilters();

The default implementation, PopupTreeFilterHeaderModel, creates a DefaultColumnFilterMapper that allows each column to be associated to either a LevelFilterMapper or a NodeFilterMapper. Therefore, you can have some columns in which the filtered node children are specifically identified and others, which filter tree nodes based on their

tree level.

Example: Use a PopupTreeFilterHeaderModel to filter the values of a TreeTableModel. All columns should show every node in the data model, except for the first column.

```
//create a (unfiltered) treetablemodel:
DefaultMutableTreeTableModel dataModel = new DefaultMutableTreeTableModel();
...fill dataModel with some values and columns

//create the FilterTreeTableModel and the tree-table
DefaultFilterTreeTableModel ftm = new DefaultFilterTreeTableModel(dataModel);
TreeTable treeTable = new TreeTable(ftm);

//create the PopupTreeFilterHeaderModel and install it to the tree-table
PopupTreeFilterHeaderModel popup = new PopupTreeFilterHeaderModel();
popup.setTableHeader((FilterTableHeader) treeTable.getTableHeader());
popup.attachToTable(treeTable);

//get the DefaultColumnFilterMapper instance
DefaultColumnFilterMapper mapper = (DefaultColumnFilterMapper) popup.
getFilterMapper();

//change the default mapping mode to NODE
mapper.setDefaultMode(DefaultColumnFilterMapper.NODE_MODE);

//make the first column show filter expressions per level
mapper.setLevel(0);
```

18 VetoableTableColumnModel

VetoableTableColumnModel is an extension to TableColumnModel that can prevent or 'veto' the addition, removal and moving of table columns from happening. Having a VetoableTableColumnModel as the table's and table header's column model can thus prove extremely useful in cases where we do not want specific columns to be removed or moved to a different location. In addition, you can prevent table columns from being added. By default, the TableColumnModels used in our component library all implement the VetoableTableColumnModel interface.

18.1 VetoableTableColumnModelListener

In order to control which columns are added, removed or moved, you should write your own **VetoableTableColumnModelListener**.
VetoableTableColumnModelListener is a listener that is notified **before** table columns are added, moved or removed. The methods that are called are:

public void columnWillBeAdded(TableColumnModelEvent e) throws ColumnModelVetoException
public void columnWillBeMoved(TableColumnModelEvent e) throws ColumnModelVetoException
public void columnWillBeRemoved(TableColumnModelEvent e) throws ColumnModelVetoException

A VetoableTableColumnModelListener can 'veto' the event by inspecting the supplied TableColumnModelEvent and fire a **ColumnModelVetoException** where appropriate. If the event is allowed to occur, a ColumnModelVetoException should **not** be thrown.

From Sun's documentation regarding TableColumnModelEvents, having **e** as the TableColumnModelEvent, bear in mind that:
for removal events: **e.getFromIndex()** identifies the column to be removed
for addition events: **e.getToIndex()** identifies the column to be added
for move events: **e.getFromIndex()** identifies the column to move from while **e.getToIndex()** the column to move to.

VetoableTableColumnModel has methods for adding and removing VetoableTableColumnModelListeners:

public void addVetoableColumnModelListener(VetoableTableColumnModelListener l)
public void removeVetoableColumnModelListener(VetoableTableColumnModelListener l)

Example 1: Veto the removal of the third table column.

```
public class MyVetoableTableColumnModelListener implements
VetoableTableColumnModelListener {
        public void columnWillBeAdded(TableColumnModelEvent e) throws
ColumnModelVetoException {
        }
        public void columnWillBeMoved(TableColumnModelEvent e) throws
ColumnModelVetoException {
        }
        public void columnWillBeRemoved(TableColumnModelEvent e) throws
ColumnModelVetoException {
                if (e.getFromIndex() == 2) throw new ColumnModelVetoException(e);
        }
}
```

Example 2: Veto the move of a column before the third column.

```
public class MyVetoableTableColumnModelListener implements
VetoableTableColumnModelListener {
        public void columnWillBeAdded(TableColumnModelEvent e) throws
```

```
ColumnModelVetoException {
        }
        public void columnWillBeMoved(TableColumnModelEvent e) throws
ColumnModelVetoException {
                if (e.getFromIndex() > 2 && e.getToIndex() <= 2) throw new
ColumnModelVetoException(e);
        }
        public void columnWillBeRemoved(TableColumnModelEvent e) throws
ColumnModelVetoException {
        }
}
```

18.2 DefaultVetoableColumnModel

DefaultVetoableColumnModel is a VetoableTableColumnModel implementation that is used throughout Citra Table. More specifically, it is created by AdvancedJTable and AdvancedTableHeader and also used by their subclasses, TreeTable, GroupTableHeader and FilterTableHeader. TreeTableHeader also uses a VetoableTableColumnModel, **TreeTableColumnModelAdapter**, which is described in its own section.

18.3 ColumnModelVetoException

ColumnModelVetoException is thrown from VetoableTableColumnModelListeners to indicate that a specific TableColumnModelEvent should **not** take place. The exception is constructed by specifying the TableColumnModelEvent and an optional message string:

public ColumnModelVetoException(TableColumnModelEvent event)
public ColumnModelVetoException(TableColumnModelEvent event, String message)

You can get the event that was responsible for the exception, with the method:

public TableColumnModelEvent getColumnModelEvent()

19 TableAssistant

TableAssistant is a class that provides additional functions to a JTable. More specifically you can:

- automatically resize the column of a table to that column's contents.
- dynamically add/remove columns through a popup menu.
- display a more detailed dialog for specifying which columns will be visible.

19.1 Creating

A TableAssistant object is created internally by AdvancedJTable. You can get an instance of it by calling AdvancedJTable's method:

public TableAssistant getTableAssistant();

You can also use this class with your own custom JTable:

```
JTable table = new JTable();
TableAssistant tableAssistant = new TableAssistant(table);
tableAssistant.register(table.getColumnModel());
```

19.2 Autoresize Table Columns

The column of a table is automatically resized to the greatest preferred width of all cells under that column, when the column is double-clicked on its border.

In addition to the column being resized upon double-clicking, you can also invoke this behaviour programmatically by calling TableAssistant's method:

public void resizeColumnToContents(int column);

19.3 Column Filter

TableAssistant provides a popup through which the columns of the table can be dynamically added/removed. The popup is shown upon right-clicking on the table header.

You can make the popup menu to show upon right-click mouse events on the header by calling TableAssistant's method:

public void setShowPopup(boolean showPopup);

You can also determine if the popup will be shown by calling:

public boolean getShowPopup();

Finally, an instance of the popup menu can be retrieved by calling:

public JPopupMenu getColumnPopup();

Example: Make the popup menu visible.

```
JPopupMenu popup = tableAssistant.getColumnPopup();
```

popup.setVisible(true);

19.4 More Dialog

If the table model contains a large number of columns, the column filter popup menu will become extremely large. For this reason, TableAssistant may display an additional dialog, where users can visually select which columns to display from a scrollable list.

You control this dialog's visibility with:

public void setShowMore(boolean showMore);
public boolean getShowMore();

Also the maximum number of columns to display in the column filter popup menu is controlled with:

public void setMaxColumns(int maxColumns);
public int getMaxColumns();

20 TableReorder

TableReorder acts on a JTable in order to ensure that the same rows are selected after the ReorderEvent is generated.

The table selections after data changes is a two-step process:
First, ReorderEvents are processed by this ReorderListener, in which step a map of how the rows have changed are stored.
Next, and after the table has finished painting the affected by the TableModelEvent area, the following method is called:

public void reselectTableRows(int[] selRows, int[] mapIndex);

, which updates the table selection.

20.1 Creating

A TableReorder object is created and managed internally by an AdvancedJTable. The following AdvancedJTable methods are used for creating, getting and setting the table reorder:

protected TableReorder createReorder();
public TableReorder getTableReorder();

You can also use this class with your own JTable subclass:

TableReorder tableReorder = new TableReorder(table);

You also need to override JTable's method

public void tableChanged(TableModelEvent e);

to look like:

```
public void tableChanged(TableModelEvent e) {
        super.tableChanged(e);
        tableReorder. reselectTableRows();
}
```

21 AdvancedTableHeader

AdvancedTableHeader extends JTableHeader in order to provide for a header that does not let column reordering when the column is being dragged with the right mouse button pressed. Additionally, you can specify which columns are allowed to be dragged and reordered.

21.1 Creating

AdvancedJTable creates and manages an AdvancedTableHeader subclass, AdvancedJTable.InnerTableHeader.

You can get an instance of the header as you would with JTable:

public JTableHeader getTableHeader();

or set it with:

public void setTableHeader(JTableHeader header);

Additionally, you can use AdvancedTableHeader in your custom JTable subclass:

```
JTable table = new JTable();
AdvancedTableHeader header = new AdvancedTableHeader(table.getColumnModel());
table.setTableHeader(header);
```

21.2 Specifying which columns can be dragged

You can control which columns can be dragged/reordered by overriding AdvancedTableHeader's

method:

public boolean isReorderingAllowed(int column);

, and return true or false accordingly.

Example: Make the second column not reorderable:

```
public boolean isReorderingAllowed(int column) {
        if (column == 1) return false;
        return true;
}
```

22 AdvancedJScrollPane

AdvancedJScrollPane has the ability to display a last column at the end of the table, and also to freeze some rows/columns at the edges of the table.

22.1 Creating

You can create and use an AdvancedJScrollPane as you would normally do with a JScrollPane:

```
AdvancedJScrollPane scroller = new AdvancedJScrollPane();
JTable table = new JTable(); //or AdvancedJTable();
scroller.setViewPortView(table);
JPanel p = new JPanel();
p.setLayout(new BorderLayout());
p.add(scroller);
```

23 Saving/loading state

The sort, filter and group states of a table or treetable can be saved and loaded anytime, by using methods of a **SortTableModel**, **FilterHeaderModel** and **DynamicTreeTableModel** respectively.

23.1 Sort state

SortTableModel contains a method for retrieving the sort state as a string:

public String getSortStatesAsString();

A string can also be used to load the sort states:

public void setSortStatesAsString(String state);

Note: In your application, you could write the sort state string to a file when exiting and load it upon initialization.

23.2 Filter state

FilterHeaderModel contains a method for serializing the filter state to an ObjectOutputStream:

public void saveFilterState(ObjectOutputStream out);

An ObjectInputStream can also be used to load the filter state:

public void loadFilterState(ObjectInputStream in);

Note: In your application, you could write the filter state to a file when exiting and load it upon initialization.

23.3 Group state

DynamicTreeTableModel and ComparableTreeTableModel contain methods for serializing the TreeTableComparators, and thus the group state, to an ObjectOutputStream:

public void saveComparators(ObjectOutputStream out);

An ObjectInputStream can also be used to load the TreeTableComparators:

public void loadComparators(ObjectInputStream in);

Note: In your application, you could write the group state to a file when exiting and load it upon initialization.

24 Searching

A table structure can be searched throughout by making use of the classes in the com.citra.table.search package. This happens by constructing a **SearchModelEvent** and have a **SearchModel** send it out to its **SearchModelListeners.** The actual searching is performed by **TableSearch**, a class which uses a Filter in order to find matching cell values. **SearchPanel** and **SearchTablePanel** are responsible for creating and propagating the search model event, and

finally, **TableSelector** acts on the event by selecting the matched table cells.

24.1 SearchModelEvent

The SearchModelEvent specifies from which cell the search will start, the search direction (forwards or backwards) and the Search object that will perform the actual searching.

In **fromRow** and **fromColumn**,
NEXT_ROW and **NEXT_COLUMN** denote the next row/column from the previously matched, while
ALL_ROWS and **ALL_COLUMNS** imply that all rows/columns of the table should be searched respectively.

24.2 Search Panels

We have created two panels through which users can search for data in a table: **SearchPanel** and **SearchTablePanel**. Both these classes construct an appropriate **SearchModelEvent** which is sent to their **SearchModelListeners**.

SearchPanel is used to find next or previous occurrences of a string which is entered inside a text field. By default, SearchPanel will search through all columns, using a (case-sensitive) StringFilter.

SearchTablePanel is also used to find next or previous of occurrences of object patterns, however, which are presented to the user via a **VisualSeeker's** panel, according to the column's java class. There are VisualSeekers for Strings, Numbers, Dates and Boolean values.

24.3 TableSearch

TableSearch performs the actual table searching via its **search(TableModel, Object, int, int, boolean)** method which returns the matching row.
The matching column can be retrieved with **getLastMatchingColumn()**. A filter is used to decide whether to accept a table's cell value. The search respects the parameters in the **search** method, but if the default column is set to anything other than **ALL_COLUMNS**, searching will only take place at the specified default column.

24.4 Example

```
//first create the search panel in order to present search options to the user
SearchTablePanel search = new SearchTablePanel(tableModel);
```

OR

```
SearchPanel search = new SearchPanel();

//now create a SearchModelListener for selecting the matched cells.
TableSelector selector = new TableSelector(table);
```

OR

```
TableStyleSelector selector = new TableStyleSelector();

//finally add selector to be notified by the search model events generated from the search
panel
search.getSearchModel().addSearchModelListener(selector);
```

25 Editors

TableCellEditors are special objects responsible for editing the cells of a JTable.
We supply two extra editors in supplement to those in Sun's JTable framework, one for editing dates and the other for presenting multiple values via a combo box.

25.1 DateEditor

DateEditor is an editor for table cells that handles the editing of Date objects. You can create a DateEditor with:

```
DateEditor de = new DateEditor();
```

AdvancedJTable installs a DateEditor upon initialization.

The editing itself is performed by a JDateChooser. JDateChooser is a JPanel that contains controls for specifying year, month, date and time.

25.2 TableComboBoxEditor

TableComboBoxEditor is an editor for table and tree table cells that uses multiple values contained in a JComboBox. TableComboBoxEditor has three constructors:

```
TableComboBoxEditor();
TableComboBoxEditor(JComboBox comboBox);
TableComboBoxEditor(String[] items);
```

Example: Create a TableComboBoxEditor.

```
String countries[] = new String[] {
        "Greece",
        "Argentina",
        "Germany",
        "Spain",
};
TableCellEditor combo = new com.citra.editors.TableComboBoxEditor(countries);
```

25.3 Setting an editor

You can set your own editor by calling JTable's following method:

```
public void setDefaultEditor(Class columnClass, TableCellEditor editor);
```

Example: Set a date editor

```
JTable table = new JTable();
DateEditor de = new DateEditor();
table.setDefaultEditor(Date.class, de);
```

Example: Set a TableComboBoxEditor editor

```
String countries[] = new String[] {
        "Greece",
        "Argentina",
        "Germany",
        "Spain",
};
TableCellEditor combo= new com.citra.editors.TableComboBoxEditor(countries);
TableColumn countryColumn;

countryColumn = table.getColumnModel().getColumn(1);
countryColumn.setCellEditor(combo);
```

26 Exporting Data

A TableModel can be easily exported to an OutputStream via the various **ExportManager** implementations. ExportManagers need to implement the method:

```
public void write(javax.swing.table.TableModel model, java.io.OutputStream out);
```

You can use **DelimitedExportManager** to write a table model in a delimited format or

XMLExportManager to create an XML document.

26.1 DelimitedExportManager

DelimitedExportManager writes a table model in a delimited format. The delimiter is specified in the constructor. There is also the option of writing the column values as the document header.

26.2 XMLExportManager

XMLExportManager writes a table model in an XML format. You should define a 'path' element in the constructor, which follows the **XPath** specification. The specified path defines parent-child node relationships from left to right, with each 'node' being separated with a **slash** (/).
e.g.
/bookstore/book
person etc.

An XML document will be created with the specified paths and the table columns as element nodes, and the cell values as the atomic values of the column nodes.

27 Internationalization

Support for localization/internationalization is provided through a property bundle file, **TableLibraryBundle.properties**. This is a text file with key values in capital letters associated to values. By modifying this file with a simple text editor, you can use your own text strings for the various UI components defined in the library.

The file **TableLibraryBundle.properties** is provided with Citra Table for the default system locale. You can supply your own property file for other locales by creating a **TableLibraryBundle_[locale].properties** file, where **[locale]** is the locale code. For example:

ru for Russian,
en for English,
en_UK for UK English,
de for German.

Make sure you publish this property file with your application.

See Sun's documentation on **java.util.ResourceBundle** for more information

27.1 Usage

TableResourceManager is a class that facilitates the retrieval of resource bundle strings. The text strings on the various UI components of our library are retrieved with TableResourceManager's method:

public static final String getString(String resourceKey);

By default, at startup, the ResourceBundle corresponding to the default locale is instantiated. However, you can also assign the ResourceBundle that will be used with the methods:

public static void setResourceBundle(ResourceBundle resource);
public static void setResourceBundle(String resource);
public static void setResourceBundle(String resource, java.util.Locale locale);

Finally, you can retrieve the current ResourceBundle with:

public static ResourceBundle getResourceBundle();

28 Renderers

TableCellRenderers are special objects responsible for rendering the cells of a JTable. We supply some extra renderers in addition to those in Sun's JTable framework.

28.1 DefaultRenderer

DefaultRenderer is the superclass of all others renderers in the com.citra.renderers package. It has methods for drawing alternative rows of a table with different color:

public void setOddColor(Color c);
public void setEvenColor(Color c);

It can also add some spaces at the end of the cell to improve the presentation of the table.

public void setSpacing(boolean spacing);
public boolean getSpacing();
public void setSpaces(int spaces);

AdvancedJTable installs a DefaultRenderer instance as the renderer for the Object class.

28.2 ProgressBarRenderer

ProgressBarRenderer is used to display the data as a progress bar value. You can specify the progress bar at construction time and retrieve it later:

```
JProgressBar bar = new JProgressBar();
ProgressBarRenderer barRend = new ProgressBarRenderer(bar);
```

or

```
ProgressBarRenderer barRend = new ProgressBarRenderer();
JProgressBar bar = barRend.getProgressBar();
```

28.3 SizeRenderer

SizeRenderer is a table cellrenderer suitable for displaying the length of a file in octets (Bytes, KB, MB, GB etc).

The octet measure's text is part of the internationalization resource bundle, **TableLibraryBundle**.

28.4 Other renderers

Renderers for all the common objects are provided.

i.e.
DateRenderer – for Dates
BooleanRenderer – for Booleans
NumberRenderer – for Numbers

28.5 Setting a renderer

There are a number of ways to to set a cell renderer.

1. Use JTable's method:

```
public void setDefaultRenderer(Class c, TableCellRenderer renderer);
```

Example:

```
TableCellRenderer cellDateRenderer = new DateRenderer();
table.setDefaultRenderer(Date.class, cellDateRenderer);
```

2. Assign a cell renderer to a column.

```
TableColumn column = table.getColumnModel().getColumn(1);
column.setRenderer(new DateRenderer());
```

3. Override the JTable's method:

```
public TableCellRenderer getCellRenderer(int row, int column);
```

to look like:

```
public TableCellRenderer getCellRenderer(int row, int column) {
        if (column == 1) {
                return cellDateRenderer;
        }
        return super.getCellRenderer(row, column);
}
```

4. Override the JTable's method:

```
public void addColumn(TableColumn aColumn);
```

to look like:

```
public void addColumn(TableColumn aColumn) {
        super.addColumn(aColumn);
        if (aColumn.getModelIndex() == 1) {
                aColumn.setCellRenderer(cellDateRenderer);
        }
}
```

29 Appendix

This section contains source code listings.

29.1 Appendix I

Source code for a ListTableModel implementation that extends javax.swing.table.DefaultTableModel

```
import javax.swing.table.*;
import java.util.*;
```

```
import com.citra.table.ListTableModel;

public class DefaultListTableModel extends DefaultTableModel implements
ListTableModel {
	public DefaultListTableModel() {
		super();
	}
	public DefaultListTableModel(java.lang.Object[][] data, java.lang.Object[]
columnNames) {
		super(data, columnNames);
	}
	public DefaultListTableModel(java.lang.Object[] columnNames, int numRows) {
		super(columnNames, numRows);
	}
	public DefaultListTableModel(int numRows, int numColumns) {
		super(numRows, numColumns);
	}
	public DefaultListTableModel(java.util.Vector columnNames, int numRows) {
		super(columnNames, numRows);
	}
	public DefaultListTableModel(java.util.Vector data, java.util.Vector
columnNames) {
		super(data, columnNames);
	}
	public void addRow(java.lang.Object row) {
		if (row instanceof Vector)
			addRow((Vector) row);
		else
			addRow((Object[]) row);
	}
	public void addRows(java.util.List addedRows) {
		for (int i = 0; i < addedRows.size(); i++) {
			addRow(addedRows.get(i));
		}
	}
	public void clear() {
		int prevsize = dataVector.size();
		dataVector.clear();

		//notify
		fireTableRowsDeleted(0, prevsize);
	}
	public Object getCellValue(Object o, int index) {
		return ((Vector) o).get(index);
	}
	public List getRows() {
		return dataVector;
	}
```

```
        public void removeRows(int[] deletedRows) {
                int len = deletedRows.length;
                int min = deletedRows[0];
                int max = deletedRows[len - 1];
                while (len-- > 0) {
                        dataVector.removeElementAt(deletedRows[len]);
                }

                //notify
                fireTableRowsDeleted(min, max);
        }
}
```

29.2 Appendix II

Source code to make cell spanning possible for a custom JTable subclass.

It is assumed that **spanDrawer** represents a SpanDrawer instance.

Override JTable's method

public Component prepareRenderer(TableCellRenderer renderer, int row, int column);

to look like:

```
public Component prepareRenderer(TableCellRenderer renderer, int row, int column) {

        Component normalComp;

        boolean useSpan = spanDrawer.getUseSpan();

        if (useSpan && spanDrawer.isCellMerged(row, column)) {
                CellSpan cs = spanDrawer.getSpanModel().getCellSpanAt(row, column);
                int spannedRow = cs.getSpannedRow();
                int spannedColumn = cs.getSpannedColumn();

                TableCellRenderer rend = getCellRenderer(spannedRow,
spannedColumn);
                normalComp = prepareRenderer(rend, spannedRow, spannedColumn);
        }
        else {
                normalComp = super.prepareRenderer(renderer, row, column);
        }

        Component ret;
        if (useSpan) ret = spanDrawer.prepare(normalComp, row, column);
```

```
            else ret = normalComp;

            return ret;
    }
```