

Article: Gantt Charts With JXTreeTable

by Ulrich Hilger, April 4, 2006

Table of Contents

Introduction	2
Scope of this article.....	2
Application scenario.....	2
Design approach	2
Implementation	3
Implementing a gantt bar component	3
GanttBar	3
Excursion: Drawing A Group Indicator	5
Interface TimeFrameProvider	5
Interface Activity	5
Implementing a suitable data model	6
Deliverable	6
ActivityTreeTableModel	6
Populating JXTreeTable with custom data.....	7
Handling data changes from JXTreeTable.....	8
Keeping parent activities synchronized	9
MutableTreeTableModel	10
Customizing date manipulation.....	10
DateCellEditor	11
Wrapping all parts into an application.....	11
GanttTreeTable	11
GanttTableCellRenderer	12
GanttDemo	13
Starting the demo	14
Conclusion.....	14
About the Author.....	15
References	15
License	15

Light Development
<http://www.lightdev.com>
info@lightdev.com

This document is available online at http://articles.lightdev.com/gantt/gantt_article.pdf

Introduction

When working on a project plan gantt¹ charts are very helpful to visualize activities within a given time frame. I recently worked on a tool for project reporting where integration with project planning was a topic so I went to look which solutions to produce project views as gantt charts are available for Java.

During research some capable items turned out to exist in the area of both project planning and charting. Some are commercial and some even are free open source products, however, all the tools are comparably big and I wondered why extending Swing with a few components would not be sufficient for my needs.

The idea was simply to use a custom cell renderer for gantt bars with a `JXTreeTable` so I spent a few hours playing with Swing and finally created a basic solution which is described in this article (see a screenshot at the end of the article).

Scope of this article

In this article it is described how to utilize the cell renderer mechanism of `JTable` and `JXTreeTable` respectively to create a gantt project plan. The article shows how to use `JXDatePicker` inside a `JXTreeTable` as a custom editor for dates too.

As a fully working demo would not be complete without some additional functionality, some common techniques such as how to build a custom data model for `JXTreeTable` or how to bind a component to application specific actions are explained as well.

Application scenario

Our demo application should support to create a project plan consisting of activities having a planned start and end date. The project plan should allow activities to be hierarchically structured and should list activities in a tabular manner. For each activity a gantt bar should visually indicate the time it occupies within a given time frame. All content should be editable easily through the user interface.

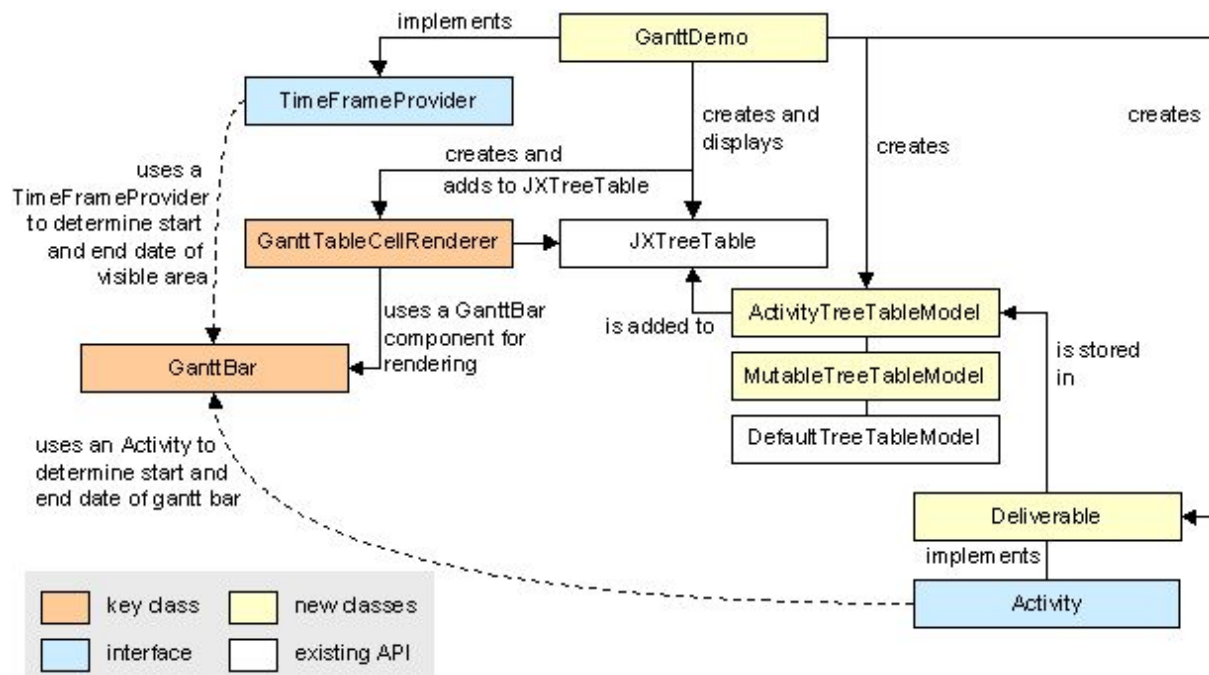
The demo application does not manage dependencies between activities, e.g. does not move activities depending on another activity when this activity is moved in time. Nevertheless such a functionality would not change the way how gantt bar rendering is implemented in this demo. It would only require an extended data model which holds references between activities and which updates dependencies in case of changes in the data model accordingly.

Design approach

Almost everything needed to build an application with the mentioned functionality already exists in the Swing package of Java [1] and the extensions from the SwingLabs project [2] respectively. The design approach in this article uses a `JXTreeTable` as the main visual component to display a project plan and `JXDatePicker` to help with date selections.

¹ Henry Laurance Gantt, 1861-1919, invented the Gantt Chart in 1910, see <http://en.wikipedia.org/wiki/Gantt>

What is missing is a way to display a gantt bar inside `JXTreeTable` for each activity. Fortunately the Swing team built a mechanism to customize the way data is displayed in `JXTreeTable` with class `TableCellRenderer`. We use this mechanism to build a custom cell renderer capable to display gantt bars. The following image shows the class model this article describes:



an overview of parts of the GanttDemo project

Although above image looks a little confusing at first sight to actually implement the mentioned functionality is very simple especially when looking at the vast functionality that already exists with components such as `JXTreeTable` or `JXDatePicker`.

As the emphasis of this article is on how to extend `JXTreeTable` with a way to display gantt charts we do not bother with persistence here and create only a “dummy” data model suitable for our demo. Please refer to “Binding JTree to a database” [3] for a basic description of how to create persistence for hierarchical data.

Implementation

In this section a step by step explanation is given for major parts of our demo application, accompanied with code snippets as appropriate. Please note that full sources are available at [5] including inline and other comments as well as a working binary. To shorten the included material in the text all comments have been removed from code snippets therefore.

Implementing a gantt bar component

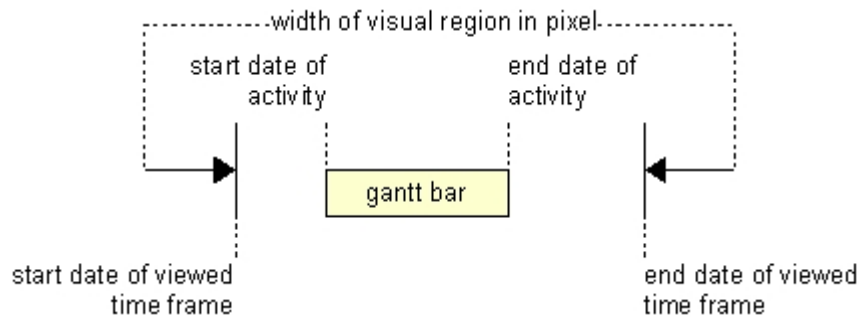
The key element of our demo application is a custom component to display a gantt bar so let us look at this component first.

GanttBar

A gantt bar is used to *visualize* the *time* an *activity* occupies inside a given *time frame*. From that definition we can derive the parts that are required to build respective component. Required are

- the start and end date of a *time frame*,
- the start and end date of an *activity* and
- the size of the region respective time frame should be *visualized* in.

The following image shows the model for our gantt bar implementation



the gantt bar model

Provided mentioned requirements are given through highlighted parts of below code snippet a gantt bar component only needs to implement a custom paint method to render itself inside a certain visual region as follows:

```
public void paint(Graphics g) {
    super.paint(g);
    if(activity != null && timeFrameProvider != null) {
        int widthInPixel = getWidth();
        double visibleStartTime = (double) timeFrameProvider.getFrameStart().getTime();
        double visibleEndTime = (double) timeFrameProvider.getFrameEnd().getTime();
        double barStartTime = (double) activity.getStart().getTime();
        double barEndTime = (double) activity.getEnd().getTime();
        double pixelDuration = (visibleEndTime - visibleStartTime) / (double) widthInPixel;
        int leftOffset = 0;
        if(visibleStartTime <= barStartTime && visibleEndTime >= barStartTime) {
            leftOffset = (int) ((barStartTime - visibleStartTime) / pixelDuration);
        }
        int rightOffset = 0;
        if(visibleStartTime <= barEndTime && visibleEndTime >= barEndTime) {
            rightOffset = (int) ((visibleEndTime - barEndTime) / pixelDuration);
        }
        g.setColor(Color.BLUE);
        g.fillRect(leftOffset, 2, widthInPixel - leftOffset - rightOffset, getHeight() - 4);
    }
}
```

method paint of class GanttBar

Using the width of the visible region method paint computes the width of a pixel inside the visible region in terms of time, i.e. which duration in the current time frame is represented by one pixel. It then finds out whether or not the left and right end of the gantt bar are located inside the visible region and if so computes their respective offsets from the boundaries of the visible region. With the resulting values it can finally draw the bar accordingly.

Class GanttBar inherits all characteristics of a component as it extends JComponent so there is hardly more to implement except for setters that actually provide the requirements for rendering.

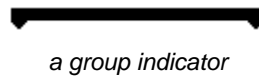
```
public void setActivity(Activity activity) {
    this.activity = activity;
}
public void setTimeFrameProvider(TimeFrameProvider timeFrameProvider) {
    this.timeFrameProvider = timeFrameProvider;
}
private Activity activity;
private TimeFrameProvider timeFrameProvider;
```

setters of class GanttBar to provide activity to render and time frame to render in

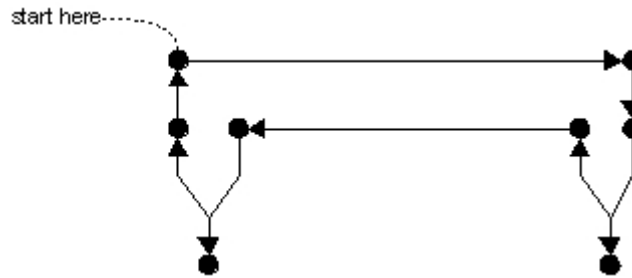
To specify start and end date of an activity as well as start and end date of a time frame to render in two interfaces are created that are named Activity and TimeFrameProvider accordingly. By using interfaces it is easy to separate the component that renders activities from other objects that might implement activities as well as a time frame to view activities in in all kinds of different ways.

Excursion: Drawing A Group Indicator

One characteristic of gantt charts is that activities can be structured hierarchically. When a hierarchy of activities applies parent activities are drawn as group indicators instead of gantt bars.



With class `Polygon` Java models a simple way to create such a group indicator and we use a `Polygon` in class `GanttBar` to draw group indicators. While the code to produce the required polygon is not so exciting (interested readers may refer to [5] instead) let us have a look at the scheme that the code implements:



polygon scheme for our group indicator

Class `Polygon` models a polygon as an array of `Points` each of which denotes the x and y coordinate of a location to change the draw direction. Class `Polygon` draws lines from one point to the next along its array of points.

Having seen how a different type of gantt bar can be drawn we come back to how to actually render different bar types depending on the data to be rendered later in the course of this article.

Interface `TimeFrameProvider`

As the name suggests interface `TimeFrameProvider` is meant to be implemented by any class that likes to provide a time frame to visualize gantt bars in. It is declared as follows:

```
public interface TimeFrameProvider {  
    public Date getFrameStart();  
    public Date getFrameEnd();  
    public static final String CMD_CHANGE_FRAME_START_DATE = "changeFrameStartDate";  
    public static final String CMD_CHANGE_FRAME_END_DATE = "changeFrameEndDate";  
}
```

interface `TimeFrameProvider`

The interface calls for only two methods `getFrameStart` and `getFrameEnd` to denote the start and end of the time frame provided. In addition it declares two constants

`CMD_CHANGE_FRAME_START_DATE` and `CMD_CHANGE_FRAME_END_DATE` which we will use later to designate commands to change the dates provided through the interface.

Interface `Activity`

Interface `Activity` needs to be implemented by any class that wishes to model characteristics of an activity having a start and end date as well as a name and a type. It is declared as follows:

```
public interface Activity {  
    public String getName();  
    public Date getStart();  
    public Date getEnd();  
    public int getType();  
    public void setName(String name);  
    public void setStart(Date start);  
}
```

```
public void setEnd(Date end);  
public void setType(int type);  
public static final int ACTIVITY_TYPE_BAR = 1;  
public static final int ACTIVITY_TYPE_GROUP = 2;  
}
```

interface Activity

The interface envisions methods `getStart`, `getEnd` and `getType` to indicate start, end and type of an activity.

From the perspective of a gantt bar component this completes what is required to render a gantt bar. We continue with building a data model for our purposes.

Implementing a suitable data model

As we now have the means to render activities we need a way to model appropriate data to provide to our component. The major data element to model a project plan in the context of our example is a activity. Please note that sources as available at [5] have parts related to data as described below in package `com.lightdev.demo.gantt.data`.

Deliverable

Class `Deliverable` models the data element that is to be manipulated and displayed in our example project plan. The class used in our demo actually is a copy from another project and implements more than needed but is a good example how different implementations can be used to model an activity. The following code snippet shows the part of class `Deliverable` relevant for our demo:

```
public class Deliverable implements Serializable, Activity {  
    public Deliverable() {  
        super();  
    }  
    ..  
    public void setStart(Date planStartDate) {  
        this.planStartDate = planStartDate;  
    }  
    public void setEnd(Date planEndDate) {  
        this.planEndDate = planEndDate;  
    }  
    public void setType(int type) {  
        this.type = type;  
    }  
    public Date getStart() {  
        return planStartDate;  
    }  
    public Date getEnd() {  
        return planEndDate;  
    }  
    public int getType() {  
        return type;  
    }  
    ..  
    private Date planStartDate = new Date();  
    private Date planEndDate = new Date();  
    private int type = 0;  
    ..  
}
```

relevant parts of class Deliverable

Among other information class `Deliverable` stores a planned start and end date. This information is passed to `GanttBar` by implementing interface `Deliverable`.

ActivityTreeTableModel

Having a class that models the data we would like to work with (class `Deliverable`) we now need a place to manage activities, a data model. Such a data model should ideally be applicable for use with `JXTreeTable` to display and manipulate stored data so we need to implement interface `TreeTableModel` with our data model. The common way to create a `TreeTableModel` is to extend

class `DefaultTreeTableModel` and implement methods as appropriate for the intended “behaviour” of the data model.

A typical behaviour of a gantt data model is to

- move an activity when its start date is changed, i.e. the end date moves along with the start date keeping the original duration
- keep the start date when the end date is changed changing the duration of the activity
- move the end date keeping the start date when the duration is changed
- adjust start, end and duration of parent activities when a child activity is changed

In the particular implementation of our demo application parent activities always reflect the minimum start and maximum end date of all child activities according to the current hierarchy of activities. Thus parent activities can not have “own” start and end dates and loose their respective settings when child activities are created for them. A way to not loose start and end dates with this implementation is to create a new activity for a designated parent activity and then move existing activities to the new parent as children.

As this data model behaviour has many aspects we look at each of them step by step.

Populating `JXTreeTable` with custom data

As data persistence is not discussed in the scope of this article, we simply create some “dummy” data initially during construction of `ActivityTreeTableModel`. See the constructor of `ActivityTreeTableModel` as well as methods `createDummyRootNode` and `createDummyData` in [5] to find out more. When interested to find out about how to implement persistence for hierarchical data article “Binding JTree to a database” [3] could point out some details.

Once data is available in the data model the central mechanism of a `TreeTableModel` to provide data to a `JXTreeTable` is method `getValueAt`. It is called automatically by `JXTreeTable` to get the values it needs to render the component.

```
public Object getValueAt(Object node, int column) {
    Object value = null;
    if(node instanceof DefaultMutableTreeNode) {
        DefaultMutableTreeNode mutableNode = (DefaultMutableTreeNode) node;
        Object o = mutableNode.getUserObject();
        if(o != null && o instanceof Activity) {
            Activity activity = (Activity) o;
            switch(column) {
                case COL_TREE:
                    value = activity.getName();
                    break;
                case COL_START:
                    value = activity.getStart();
                    break;
                case COL_END:
                    value = activity.getEnd();
                    break;
                case COL_DAYS:
                    double startInMilliseconds = (double) activity.getStart().getTime();
                    double endInMilliseconds = (double) activity.getEnd().getTime();
                    value = (endInMilliseconds - startInMilliseconds) / MILLISECONDS_IN_A_DAY;
                    break;
            }
        }
    }
}
```

```
        case COL_GANTT:
            value = activity;
            if (mutableNode.isLeaf()) {
                ((Activity) value).setType(Activity.ACTIVITY_TYPE_BAR);
            }
            else {
                ((Activity) value).setType(Activity.ACTIVITY_TYPE_GROUP);
            }
            break;
        }
    }
}
return value;
}
```

method getValueAt of class ActivityTreeTableModel

In parameter node method `getValueAt` is passed the tree node a value is to be determined from. Our implementation of method `getValueAt` first retrieves the user object from that tree node and casts it to a `Activity`. We now can access contents of the found activity according to the desired column as passed in parameter `column`.

While for other columns simply a field from respective activity object is returned, a dynamically computed value is returned for column `COL_DAYS`. The method computes the number of days between plan start and end date from the activity and returns it as the duration of the activity.

A reference to the entire activity object is returned for `COL_GANTT` which is a mechanism we come back to in greater detail later. Please note however, that the type of gantt bar (bar or group indicator) is determined in `ActivityTreeTableModel` from the type of node (leaf or parent).

Handling data changes from JXTreeTable

Method `setValueAt` of the data model associated to a `JXTreeTable` is called automatically whenever changes were made in a `JXTreeTable`. Consequently method `setValueAt` implements all previously described custom gantt data model behaviour in our `ActivityTreeTableModel`.

```
public void setValueAt(Object aValue, Object node, int column) {
    if (node instanceof DefaultMutableTreeNode) {
        DefaultMutableTreeNode mutableNode = (DefaultMutableTreeNode) node;
        Object o = mutableNode.getUserObject();
        if (o != null && o instanceof Activity) {
            Activity activity = (Activity) o;
            switch (column) {
                case COL_TREE:
                    activity.setName(aValue.toString());
                    break;
                case COL_START:
                    Date newStartDate = (Date) aValue;
                    double startInMilliseconds = (double) activity.getStart().getTime();
                    double newStartInMilliseconds = (double) newStartDate.getTime();
                    double differenceInMilliseconds = newStartInMilliseconds - startInMilliseconds;
                    double endInMilliseconds = (double) activity.getEnd().getTime();
                    double newEndInMilliseconds = endInMilliseconds + differenceInMilliseconds;
                    activity.setStart(newStartDate);
                    activity.setEnd(new Date((long) newEndInMilliseconds));
                    updateNode((DefaultMutableTreeNode) mutableNode.getParent());
                    break;
                case COL_END:
                    activity.setEnd((Date) aValue);
                    updateNode((DefaultMutableTreeNode) mutableNode.getParent());
                    break;
            }
        }
    }
}
```



```
        case COL_DAYS:
            startInMilliseconds = (double) activity.getStart().getTime();
            newEndInMilliseconds =
                startInMilliseconds + ((Double) aValue).doubleValue() * MILLISECONDS_IN_A_DAY;
            activity.setEnd(new Date((long) newEndInMilliseconds));
            updateNode((DefaultMutableTreeNode) mutableNode.getParent());
            break;
        case COL_GANTT:
            break;
    }
}
```

method setValueAt of class ActivityTreeTableModel

In parameter node method `setValueAt` is passed the tree node a changed value is to be stored for. Our implementation of method `setValueAt` first retrieves the user object from that tree node and casts it to an `Activity`. We can now access the activity object to change the value designated by parameter `column`. The new value for respective column is passed in parameter `aValue`.

While for other columns simply `aValue` is stored in the activity as new value, columns `COL_START` and `COL_DAYS` have the parts that implement the typical gantt data model behaviour.

When the start date has been changed (as reflected by paramter `column` having value `COL_START`), method `setValueAt` computes the number of days the start date is different from its original value and adjusts the activity's end date along with the start date accordingly. Similarly when the number of days was changed it is added to the current start date and the resulting new end date is stored in the affected activity.

Note that for columns `COL_START`, `COL_END` and `COL_DAYS` method `updateNode` is called in addition which we come to in the next section.

Keeping parent activities synchronized

As previously described one "behaviour" of `ActivityTreeTableModel` should be that activities may have a hierarchical structure in which parent activities shall always reflect earliest start, latest end and total duration over all child activities.

Method `setValueAt` only stores changes to the currently edited row of `JXTreeTable` so an additional method is required which we can plug in to all relevant parts of method `setValueAt` for our needs. Whenever start, end or duration of a activity is changed, method `updateNode` is called in addition. Method `updateNode` is implemented as follows:

```
private void updateNode(DefaultMutableTreeNode parentNode) {
    if(parentNode != null && !parentNode.isLeaf()) {
        Activity activity = (Activity) parentNode.getUserObject();
        long startDate = activity.getStart().getTime();
        long endDate = activity.getEnd().getTime();
        long newStart = Long.MAX_VALUE;
        long newEnd = Long.MIN_VALUE;
        Enumeration children = parentNode.children();
        while(children.hasMoreElements()) {
            DefaultMutableTreeNode child = (DefaultMutableTreeNode) children.nextElement();
            Activity childD = (Activity) child.getUserObject();
            newStart = Math.min(newStart, childD.getStart().getTime());
            newEnd = Math.max(newEnd, childD.getEnd().getTime());
        }
        if(newStart != startDate || newEnd != endDate) {
            activity.setStart(new Date(newStart));
            activity.setEnd(new Date(newEnd));
        }
        if(!parentNode.isRoot()) {
            updateNode((DefaultMutableTreeNode) parentNode.getParent());
        }
    }
}
```

method setValueAt of class ActivityTreeTableModel

Method `updateNode` is passed a node of `JXTreeTable` which the method assumes to be the parent node of a recently changed node. For this parent node the user object is retrieved and casted to a `Activity` object.

The method then iterates over all child nodes of the given parent and determines the earliest start and latest end date among all children. When earliest start or latest end date differ from the parent's current setting the activity of that parent is updated with the new dates accordingly.

Finally method `updateNode` finds out whether or not the current parent is the root node (the topmost node in the hierarchy of activities). If not, it calls itself with its parent as a parameter, this way going all the way up the tree hierarchy recursively adjusting all start and end dates as appropriate.

MutableTreeTableModel

Before leaving our data model implementation one notable fact not mentioned so far is that `ActivityTreeTableModel` does not extend `DefaultTreeTableModel` directly. Instead we built in an additional class `MutableTreeTableModel` which extends `DefaultTreeTableModel` and which in turn is extended by `ActivityTreeTableModel`.

Later in the course of this article we apply some data manipulation functionality using methods which originally come from `DefaultTreeModel`. Although `DefaultTreeModel` is part of the Java Runtime Environment (JRE) and as such is available to our application we can not use the methods from `DefaultTreeModel` due to inheritance restrictions in the implementation of `JXTreeTable`. Instead we have to copy two methods `insertNodeInto` and `removeNodeFromParent` from `DefaultTreeModel` into our `MutableTreeTableModel` (see [5]).

Customizing date manipulation

To deal with activities in a project plan involves a lot of date entries which are comparably hard to manipulate when only a date string can be edited. Due to the many possible date formats and input restrictions editing a date string becomes cumbersome in most cases. In the context of our demo application two areas involve date manipulation:

- To change start or end date of the time frame viewed and
- to change start or end date of a activity inside the `JXTreeTable`.

Component `JXDatePicker` from the SwingLabs project [2] is used to simplify date manipulation in our demo. For changes to start or end date of the time frame viewed simply two `JXDatePicker` components are added to our demo application (see also class `GanttDemo` later in this article).

```
private void buildUi() {
    ..
    JPanel timeFrameSelector = new JPanel();
    JLabel lb = new JLabel("Start:");
    timeFrameSelector.add(lb);
    timeFrameStart = new JXDatePicker();
    timeFrameSelector.add(timeFrameStart);
    lb = new JLabel("End:");
    timeFrameSelector.add(lb);
    timeFrameEnd = new JXDatePicker();
    timeFrameSelector.add(timeFrameEnd);
    contentPane.add(timeFrameSelector, BorderLayout.NORTH);
    ..
    timeFrameStart.setActionCommand(TimeFrameProvider.CMD_CHANGE_FRAME_START_DATE);
    timeFrameEnd.setActionCommand(TimeFrameProvider.CMD_CHANGE_FRAME_END_DATE);
    ..
    timeFrameStart.addActionListener(treeTable);
    timeFrameEnd.addActionListener(treeTable);
    ..
}
..
private JXDatePicker timeFrameStart;
private JXDatePicker timeFrameEnd;
..
```

parts from method `buildUi` of class `GanttDemo` to use `JXDatePicker` for editing the time frame viewed

In the relevant parts of method `buildUi` of class `GanttDemo` two `JXDatePicker` objects are instantiated and added to the application frame. The date picker objects each are associated with an action command for identification and our `JXTreeTable` object is added as an `ActionListener` to be notified of date changes.

DateCellEditor

To allow for custom date manipulation using `JXDatePicker` objects on date fields inside a `JXTreeTable` a custom cell editor is required. For the purpose of our demo application we create class `DateCellEditor` as follows:

```
public class DateCellEditor extends AbstractCellEditor implements TableCellEditor {
    public DateCellEditor() {
        datePicker = new JXDatePicker();
        datePicker.setFormats(new DateFormat[] { DateFormat.getDateInstance(DateFormat.SHORT) });
    }
    public Object getCellEditorValue() {
        return datePicker.getDate();
    }
    public Component getTableCellEditorComponent(
        JTable table, Object value, boolean isSelected, int row, int column)
    {
        if(value != null && value instanceof Date) {
            datePicker.setDate((Date) value);
        }
        return datePicker;
    }
    private JXDatePicker datePicker;
}
```

a custom cell editor for date manipulation utilizing JXDatePicker

Class `DateCellEditor` creates a `JXDatePicker` upon construction and implements interface `TableCellEditor` to return that `JXDatePicker` in method `getTableCellEditorComponent`. Method `getCellEditorValue` is used to return the current date from the `JXDatePicker`. Our cell editor is created in method `buildUi` of class `GanttDemo` too:

```
private void buildUi() {
    ..
    DateCellEditor dce = new DateCellEditor();
    treeTable.setDefaultEditor(Date.class, dce);
    ..
}
```

creation and usage of a custom cell editor for date manipulation

After creation `DateCellEditor` is associated as the default editor for fields of type `Date` in our instance of `JXTreeTable`.

Wrapping all parts into an application

After major parts of our demo have been discussed it is left to look at how all parts are put together to form a real application. Again only key parts are discussed here while all parts are shown in [5] in greater detail.

Before looking at the remaining parts of application class `GanttDemo` itself we have a look at how we get `JXTreeTable` to handle events related to our gantt chart and see how our custom gantt bar component is utilized by a custom renderer.

GanttTreeTable

We want our tree table for gantt charts to handle actions related to the displayed chart, i.e. manipulation of start and end date of time frame viewed as well as creation and deletion of activities. A simple way to achieve this is to extend `JXTreeTable` with an own class and let that class implement interface `ActionListener`. With a new subclass of `JXTreeTable` it can also be assured that the appropriate data model, renderer and editor is used through the use of a custom constructor. With class `GanttTreeTable` we do this as follows:

```
public class GanttTreeTable extends JXTreeTable implements ActionListener {
    public GanttTreeTable(ActivityTreeTableModel treeModel, GanttTableCellRenderer gr,
        DateCellEditor dateEditor)
    {
        super(treeModel);
        setDefaultRenderer(GanttBar.class, gr);
        setDefaultEditor(Date.class, dateEditor);
    }
    private void repaintTreeTable() {
        invalidate();
        repaint();
    }
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if(cmd.equals(CMD_NEW_ACTIVITY)) {
            createInSelectedRow();
        }
        else if(cmd.equals(CMD_DELETE_ACTIVITY)) {
            deleteSelectedRow();
        }
        else if(cmd.equals(TimeFrameProvider.CMD_CHANGE_FRAME_START_DATE)) {
            repaintTreeTable();
        }
        else if(cmd.equals(TimeFrameProvider.CMD_CHANGE_FRAME_END_DATE)) {
            repaintTreeTable();
        }
    }
    ..
    public static final String CMD_NEW_ACTIVITY = "newActivity";
    public static final String CMD_DELETE_ACTIVITY = "deleteActivity";
}
```

class GanttTreeTable

In method `actionPerformed` `GanttTreeTable` uses the action command found in the event object to determine which action is to be performed. It handles the event mainly to repaint itself after a change or to call to create or delete a activity.

As already shown in chapter “Customizing date manipulation” class `GanttTreeTable` can be associated with the relevant components that produce action events affecting the gantt chart.

```
private void buildUi() {
    ..
    newBtn.setActionCommand(GanttTreeTable.CMD_NEW_ACTIVITY);
    deleteBtn.setActionCommand(GanttTreeTable.CMD_DELETE_ACTIVITY);
    ..
    newBtn.addActionListener(treeTable);
    deleteBtn.addActionListener(treeTable);
    ..
}
```

associating GanttTreeTable with other components as an ActionListener

GanttTableCellRenderer

In the discussion of method `getValueAt` is described how `JXTreeTable` determines *what* needs to be rendered in a particular table cell. To find out about `JXTreeTable` determining *how* to render a table cell we look at the `TableCellRenderer` mechanism here.

With class `GanttTableCellRenderer` a class is created that associates our `GanttBar` component with a `JXTreeTable` to render activities as gantt bars.

```
public class GanttTableCellRenderer extends DefaultTableCellRenderer {
    public GanttTableCellRenderer(TimeFrameProvider timeFrameProvider) {
        super();
        bar = new GanttBar();
        bar.setTimeFrameProvider(timeFrameProvider);
    }
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column)
    {

```

```
    setValue(value);  
    return bar;  
}  
protected void setValue(Object value) {  
    if(value != null && value instanceof Activity) {  
        bar.setActivity((Activity) value);  
    }  
}  
private GanttBar bar;  
}
```

class GanttTableCellRenderer

Class `GanttTableCellRenderer` extends class `DefaultTableCellRenderer` and creates an instance of a `GanttBar` object in its constructor. It associates a `TimeFrameProvider` with the `GanttBar` as received in respective parameter. Whenever `JXTreeTable` encounters a class that is associated with `GanttTableCellRenderer`, `JXTreeTable` automatically calls method `getTableCellRenderer` of class `GanttTableCellRender`. To associate our renderer with `JXTreeTable` in this way, again method `buildUi` of class `GanttDemo` is used:

```
private void buildUi() {  
    ..  
    treeTable.setDefaultRenderer(GanttBar.class, new GanttTableCellRenderer(this));  
    ..  
}
```

associate a GanttBar column with a GanttTableCellRenderer

Class `ActivityTreeTableModel` ensures that our renderer is used by `JXTreeTable` with method `getColumnClass` where it returns class `GanttBar` accordingly (see [5]).

`GanttTableRenderer` expects an object that implements interface `Activity` in parameter value of method `getTableCellRendererComponent`. It passes this `Activity` instance on to the `GanttBar` object and returns the `GanttBar` object as the component to use for rendering.

The trick now is to build a bridge between the activities stored in the data model and the `GanttBar` instance that does the actual rendering of activities. This is achieved in already mentioned method `getValueAt` of class `ActivityTreeTableModel` which returns a `Activity` object for the gantt bar column. Class `Activity` implements interface `Activity` and can be used by `GanttBar` as described.

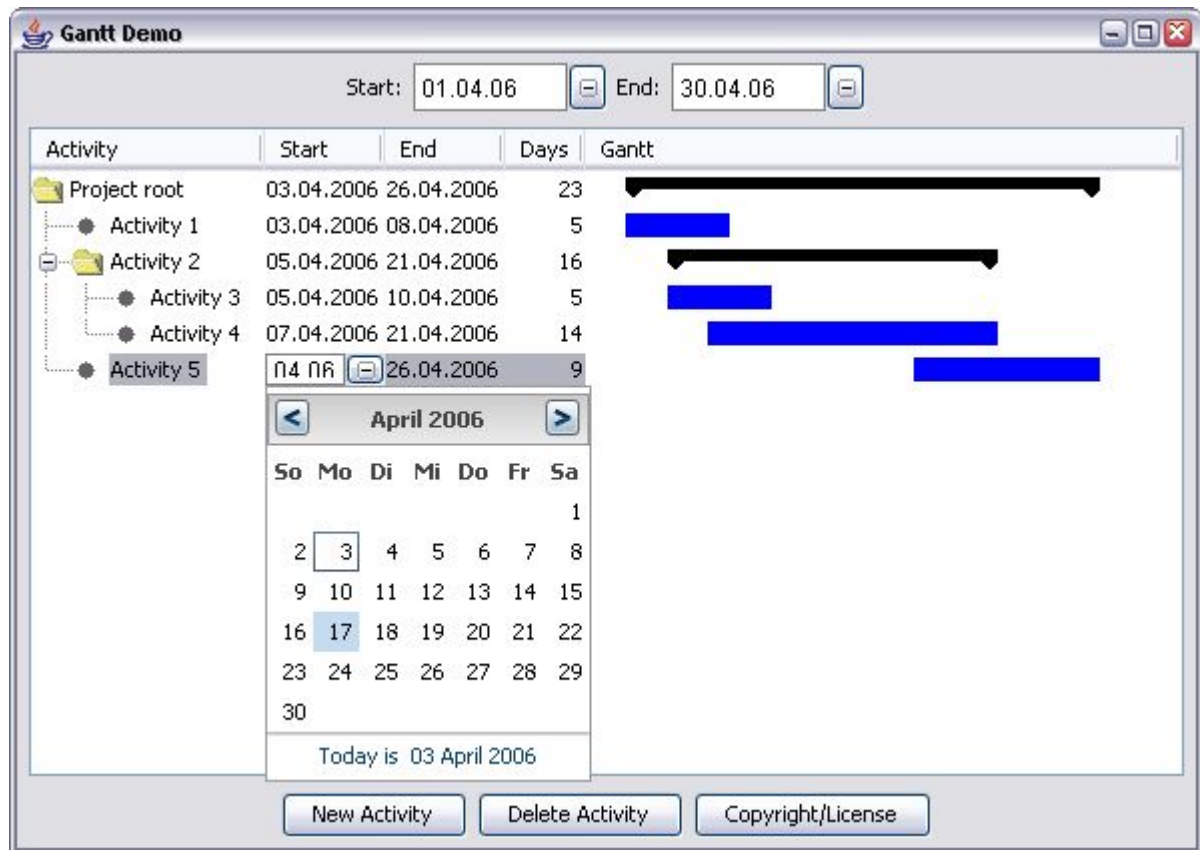
GanttDemo

With class `GanttDemo` finally an application is built that puts together all described parts in more or less one central method `buildUi`. As the name suggests method `buildUi` builds a user interface by instanciating the many classes involved to produce a gantt chart and bringing those objects together on a `JFrame`. Most major aspects of how this is done are explained already previously so the rest is omitted for being looked at in greater detail individually at [5].

In method `buildUi` drag and drop is established too as an adaption of what has been described in detail in "Extending JTree capabilities" [4] which is why it is not described here either.

Starting the demo

Included in [5] is an executable Java archive file named `GanttDemo.jar` which was built from the included sources. Simply double click on the icon of that file to start the demo application. Or you can Web Start it online at [8]. Once the demo started it should produce a window such as the following:



screenshot of GanttDemo application

You can play with the demo data by dragging and dropping activities around, changing start and end dates of activities or the time frame viewed as well as by using the buttons beneath the gantt chart to create or delete activities.

Conclusion

Utilizing a custom table cell renderer mechanism it is simple to extend existing Java functionality to visualize and manipulate a project plan. The more complex part is to find, choose, extend and orchestrate all the many parts of existing APIs for own requirements for which this article presents a straightforward and pragmatic solution approach.

In cases when only a quick view or manipulation of project plan data is required, existing Java components can be used to do the job fast and efficiently being an alternative to the full monty of an entire project planning system.

Another advantage of using Java as shown here is that all kinds of different data models can be visualized and manipulated with the presented approach regardless of data being stored locally or remotely as opposed to the proprietary data formats mostly connected to the use of project planning systems.

Still the solution shown in this article can be extended to add more features such as dependencies between activities, a custom gantt column headline with variable date sequences or support for data persistence [3] for instance.

About the Author

Ulrich Hilger does software development based on more than 25 years of experience in the software industry. In his spare time he creates software solutions which he publishes under the label Light Development [7]. Platform independent solutions based on Java technology are the main activity sector since 1999.

References

- [1] Java 5 Standard Edition Software Development Kit (JDK 5)
<http://java.sun.com/j2se/1.5.0/>
- [2] SwingLabs project
<http://swinglabs.org/>
- [3] Article "Binding JTree to a database"
http://articles.lightdev.com/boundtree/boundtree_article.pdf
- [4] Article "Extending JTree capabilities"
http://articles.lightdev.com/tree/tree_article.pdf
- [5] Source codes and binary of the GanttDemo application
<http://articles.lightdev.com/gantt/GanttDemo.zip>
- [6] This article on the web
http://articles.lightdev.com/gantt/gantt_article.pdf
- [7] Light Development - Ulrich Hilger's web site
<http://www.lightdev.com>
- [8] Web Start GanttDemo online
<http://articles.lightdev.com/gantt/GanttDemo.jnlp>

License

Copyright (c) 2006 Ulrich Hilger, Light Development, <http://www.lightdev.com>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of this document and accompanying source codes must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Light Development nor the names of its contributors may be used to endorse or promote products derived from this document and accompanying software without specific prior written permission.

THIS DOCUMENT AND ACCOMPANYING SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT AND ACCOMPANYING SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.