JavaGantt 2011.1

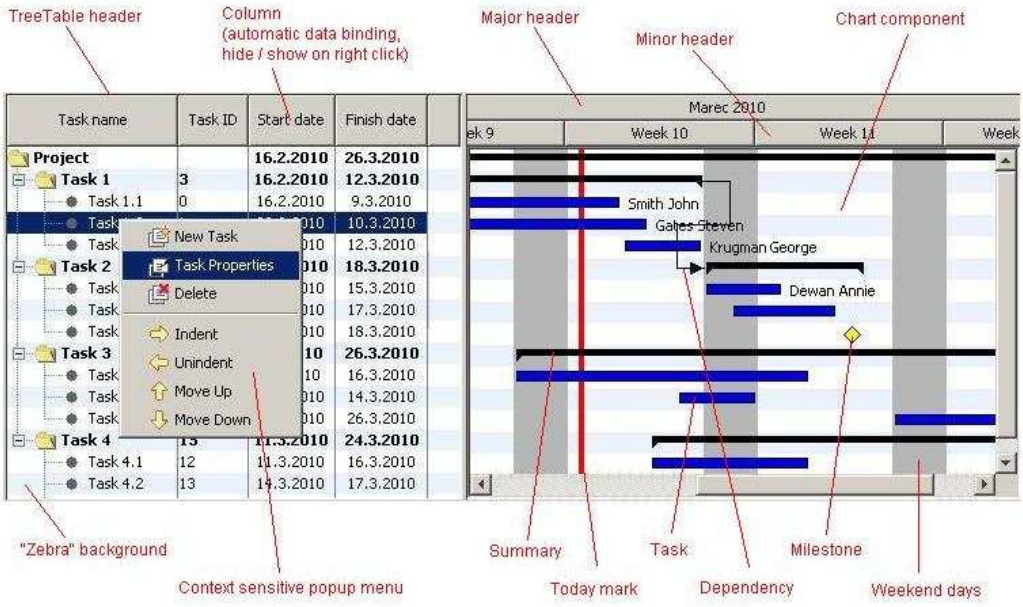
Manual

www.beesoft.eu

Content

1 Introduction

JavaGantt is the Swing component for the Java gantt chart painting and editing.



It is powerful component, despite its complexity focused on easy-to-use programming. You can see it has the *"standard gantt chart"* look. But almost all you see can be customized. Your chart can look quite differently. This manual can help you to teach how you can do it.

1.1 JavaGantt features

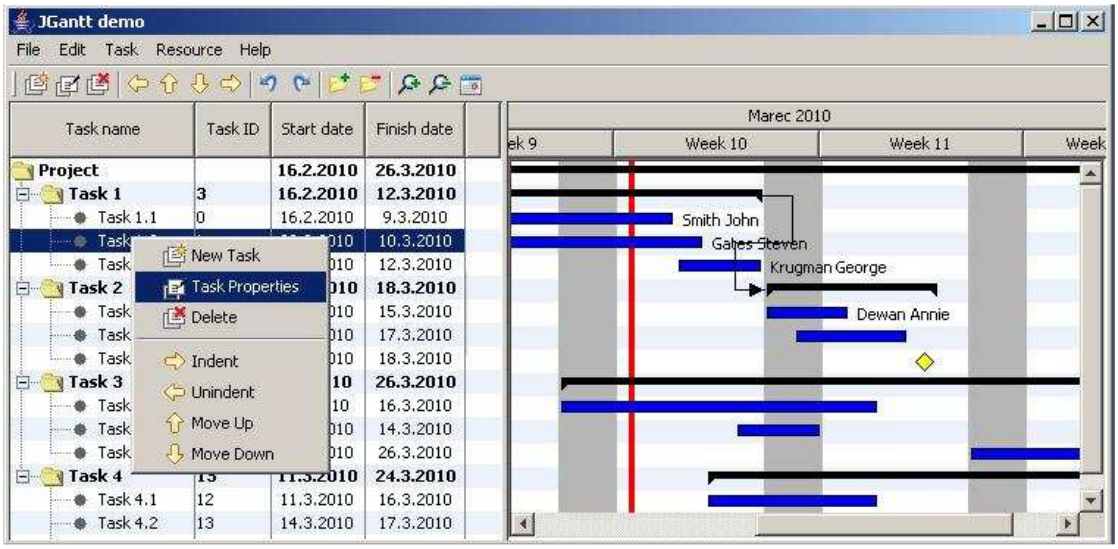
This page contains descriptions of JavaGantt's most essential features:

- **Standard view** - Standard chart appearance and behavior customizable settings. There are a treetable and a chart component side by side. The treetable on the left hand side, and the chart component on the right. Exactly as you know it from other gantt components. But you can modify their appearance as you need.
- **Advanced treetable** - The treetable has a few very interesting features: It is derived from *eu.beesoft.gaia.swing.JTreeTable*, so it has the features of JTable and JTree - it is easy to use it, if you are familiar with Swing programming. JTreeTable can paint *"zebra"* background and also show / hide columns on user selection (right click on treetable header displays popup menu with column names to hide / show). There is also built-in support for the context popup menu and double click processing.

- **Data binding** - Model for JavaGantt is oriented on the easy programming and usage. One of its features is automated data binding. You can set a name of the property of your domain object you want to display in some column and JavaGantt displays it without any more programming. And the same it does with setting user-changed visual data to domain object.
- **Lazy loading** - If your application works with a large dataset, it is suitable to load to the memory just the visible part of the data. JavaGantt treetable listens to treeexpand event and if node is expanded for the first time, it invokes method *explore()* on *GanttModel*. This is the place you can load data for node's user object and build subnodes. Do you know any more simple solution?
- **Layers** - Gantt chart is painted in some layers. You can choose which layers you want install into gantt chart, and also you can hide / show layers at runtime. Each layer has specific purpose and paints only the gadgets it is programmed for. You can develop new layers or customize the existing. This is the way you can customize the whole chart, change the shape of timeline objects or dependency lines or add an absolutelly new functionality. In each layer you can solve also a mouse service, there is a built-in support.
- **Relationships** - Timeline objects (tasks) can have relationship with each other. You can define constraints on such relationship (for example: this task can start after that task is finished). In the JavaGantt you need implement one interface on the domain objects level. And you can subclass layer for dependencies painting to get your own visualization of dependencies, if the delivered is not suitable for you.
- **Actions** - JavaGantt comes with predefined set of actions. There are actions for zoom-in and zoom-out, creating, deleting and moving nodes (and their user-objects) in the tree hierarchy, and also actions for undo and redo operations. And also abstract superclasses for the smart building another actions.
- **Localization** - JavaGantt uses data from resource bundle for treetable column names and action properties. There is a built-in mechanism to do it without any programming. If your application contains more resource bundles for more languages, you can build a multi-lingual gantt chart in a very simple way. JavaGantt listens to the language changes at runtime and repaints itself if gets an event.

1.2 Quick start

Here is the picture of our demo application distributed with JavaGantt:



The source codes are distributed as examples with JavaGantt, too. All of it you can find in DEMO directory in our distribution. Let's see what you have to do to build your own application based on JavaGantt - in 9 java files with about 40 KB texts.

Every object you want to display in JavaGantt treetable must implement *eu.beesoft.gantt.TimelineObject* interface. There are methods in this interface to get / set a start and end date of object, etc. In our example we create class *Task* for our domain objects. Our *Task* is the bean with some additional logic to handle start and end date changes:

```
public class Task extends AbstractBean implements TimelineObject {
     private String taskId;
     private String name;
     private String description;
     private Date startDate;
     private Date endDate;
     private long effort;
     private boolean milestone;
     private Task supertask;
     private List<Task> subtask;
     private List<Dependency> dependencies;
     private Resource assignee;


     public Task () {
            // empty
     }


     public String getName () {
            return name;
     }


     public void setName (String name) {
            String old = this.name;
            this.name = name;
```

```
            firePropertyChange ("name", old, name);
    }

       .....

    public Date getStartDate () {
            if ((startDate == null) && isSummary ()) {
                    updateStart ();
            }
            return startDate;
    }


    public void setStartDate (Date start) {
            if (Objects.equals (this.startDate, start)) {
                    return;
            }
            Date old = this.startDate;
            this.startDate = start;
            if (supertask != null) {
                    supertask.updateStart ();
            }
            firePropertyChange ("startDate", old, start);
    }


    private void updateStart () {
            if (isSummary ()) {
                    Date thisStart = null;
                    List<Task> tasks = getSubtask ();
                    for (Task task : tasks) {
                            Date taskStart = task.getStartDate ();
                            if (taskStart != null) {
                                    if ((thisStart == null) ||
                                      thisStart.after (taskStart)) {
                                            thisStart = taskStart;
                                    }
                            }
                    }
                    if (thisStart != null) {
                            setStartDate (thisStart);
                    }
            }
    }

       .....
}
```

Next, we create our model. We implement these methods:

- *getColumnClass()* - in our treetable are dates in columns 2 and 3 (counted from 0), so we want to say to the treetable about it
- *isCellEditable()* - column 1 contains non-editable task ID, other columns are editable
- *explore()* - method is invoked when treenode is the first time expanded and has no children yet. In our example we build new nodes for task children.

- *moveObject()*, *deleteObject()* and *createObject()* - these methods are required to implement by parent class (they are abstract). They are invoked when user executes corresponding action. JavaGantt makes its work on treetable nodes level, but a programmer has to serve the domain objects level. So in these methods we handle the *Task* objects hierarchy. Note the undo / redo support you should use in these methods.
- *getLabel()* - we use the LabelLayer layer later in our example and it requires to implement this method - here it returns a name of a task assignee

Here is the source:

```
public class Model extends GanttModel {

	public Model () {
		super ();
	}



	@Override
	public Class getColumnClass (int column) {
		switch (column) {
			case 2:
			case 3:
				return Date.class;
			default:
				return super.getColumnClass (column);
		}
	}


	@Override
	public boolean isCellEditable (TreeTableNode node, int column) {
		if (column == 0) {
			return true;
		}
		return (column >= 2);
	}


	@Override
	public void explore (GanttNode node) {
		Task task = (Task) node.getUserObject ();
		for (Task t : task.getSubtask ()) {
			GanttNode child = new GanttNode (this, t);
			if (t.getSubtask ().isEmpty ()) {
				child.setExplorable (false);
			}
			node.add (child);
		}
	}


	@Override
	public boolean moveObject (TimelineObject object,
		TimelineObject newContainer, int newIndex, UndoStep undo)
{
		Task task = (Task) object;
```

```
            Task supertask = (Task) newContainer;
            undo.registerObject (task);
            undo.registerObject (supertask);
            Task oldSupertask = task.getSupertask ();
            if (supertask != oldSupertask) {
                    if (oldSupertask != null) {
                            undo.registerObject (oldSupertask);
                            oldSupertask.removeSubtask (task);
                    }
                    supertask.addSubtask (task);
            }
            supertask.shiftSubtask (task, newIndex);
            return true;
    }


    @Override
    public boolean deleteObject (TimelineObject what, UndoStep undo)
{
            Task task = (Task) what;
            Task parent = task.getSupertask ();
            undo.registerObject (parent);
            undo.registerObject (task);
            parent.removeSubtask (task);
            return true;
    }


    @Override
    public TimelineObject createObject (TimelineObject parent, int
index,
             UndoStep undo) {
            undo.registerObject (parent);
            Task parentTask = (Task) parent;
            Task task = new Task ();
            String id = Gantt.getNextTaskId ();
            task.setTaskId (id);
            task.setName ("Task " + id);
            parentTask.addSubtask (task);
            parentTask.shiftSubtask (task, index);
            return task;
    }


    @Override
    public String getLabel (TimelineObject object) {
            if (object instanceof Task) {
                    Resource assignee = ((Task) object).getAssignee
();
                    if (assignee != null) {
                            return assignee.getName ();
                    }
            }
            return null;
    }
}
```

Now we can create our *DemoGantt* class that subclasses JavaGantt. It is very simple class, it has only two methods to support mouse actions:

- *getPopupActions()* - returns the list of actions to show in popup menu
- *doubleClicked()* - invokes dialog to edit task properties when user double-clicked on task node

```
public class DemoGantt extends JavaGantt {

     private static Action createNodeAction;
     private static Action editNodeAction;
     private static Action deleteNodeAction;
     private static Action indentNodeAction;
     private static Action unindentNodeAction;
     private static Action moveNodeUpAction;
     private static Action moveNodeDownAction;
     private static Action expandAllNodesAction;
     private static Action collapseAllNodesAction;
     private static Action zoomInAction;
     private static Action zoomOutAction;
     private static Action todayAction;
     private static Action undoAction;
     private static Action redoAction;
     private List popupActions;


     public DemoGantt (GanttModel model) {
            super (model);
     }


     @Override
     protected List<Action> getPopupActions (List<GanttNode>
selectedNodes) {
            if (popupActions == null) {
                   popupActions = new ArrayList<Action> ();
                   popupActions.add (createNodeAction);
                   popupActions.add (editNodeAction);
                   popupActions.add (deleteNodeAction);
                   popupActions.add (null);
                   popupActions.add (indentNodeAction);
                   popupActions.add (unindentNodeAction);
                   popupActions.add (moveNodeUpAction);
                   popupActions.add (moveNodeDownAction);
            }
            return popupActions;
     }


     @Override
     public void doubleClicked (GanttNode node) {
            Task task = (Task) node.getUserObject ();
            TaskPropertiesAction.showTaskProperties (task, this);
     }
}
```

To keep the example here as simple as possible we ignore another auxiliary classes (as actions, form to edit task properties or date cell renderer). All of them you can find in our DEMO directory.

So what we need is put it all together. It is done in *DemoGantt.main()* method:

```
// create model
// it extends GanttModel
GanttModel model = new Model ();

// create domain objects
// they should come from database
// or server in real application
Resource worker_1 = new Resource ();
worker_1.setName ("Smith John");

.....

Task task_1_1 = createTask ("Task 1.1", -20, +1, 92);
task_1_1.setAssignee (worker_1);

.....

Task task_1 = createTask ("Task 1", 0, 0, 0);
task_1.addSubtask (task_1_1);

.....

Project project = new Project ();
project.setName ("Project");
project.addResource (worker_1);
project.addSubtask (task_1);

.....

// set created project as gantt model root object
model.setRootObject (project);


.....

// create gantt component with "zebra" background
// and vertical column-line-separators
JavaGantt gantt = new DemoGantt (model);
JTreeTable treeTable = gantt.getTreeTable ();
treeTable.setAlternateBackground (treeTable
        .getDefaultAlternateBackground ());
treeTable.setShowVerticalLines (true);

// create columns in gantt tree-table
// this is the first column with task tree
GanttColumn column = new GanttColumn ();
column.setHeaderValue ("Task name");
column.setBinding ("name");
gantt.addColumn (column);

column = new GanttColumn ();
column.setHeaderValue ("Task ID");
column.setBinding ("taskId");
gantt.addColumn (column);

.....

// resize columns to appropriate width
```

```
            gantt.getTreeTable ().pack ();

            // create layers for gantt chart
            // order of layers is important for painting
            gantt.addLayer (new BackgroundLayer ());
            gantt.addLayer (new CalendarLayer ());
            gantt.addLayer (new GridLayer ());
            gantt.addLayer (new TodayLayer ());
            gantt.addLayer (new GanttNodeLayer ());
            gantt.addLayer (new LabelLayer ());
            gantt.addLayer (new DependencyLayer ());

            // create gantt actions
            createNodeAction = new CreateNodeAction (gantt);
            editNodeAction = new TaskPropertiesAction (gantt);

            .....

            build menu and toolbar, create a swing frame and show it
```

Note in this example:

- you don't need to act with treenodes - we build our task hierarchy and then we set the top object (*project*) as the root object to model
- note *setBinding()* method usage when we construct treetable columns - by this method you can tell to the treetable which property of your domain object you want to display in the column - there is no manual programming of data get / set access
- note also layers setting - you can see that the look of the chart component is conditional - it depends on the layers you use to paint it. You can write your own layers to give your application a specific look.

2 JavaGantt installation

Simply unzip downloaded file to appropriate directory (you must create such directory). You will get this structure:

```
JAVAGANTT                       - this is the root directory
    license.txt                 - license agreement
    release.txt                 - release notes
    DEMO                        - source examples
    DOC                         - documentation directory
         javagantt_2011_1.pdf   - JavaGantt manual
         API                    - Javadoc directory
    LIB                         - jars directory
         javagantt_2011_1.jar   - component library
         gaia_2011_2.jar        - BeeSoft general library
         launcher.jar           - library for launch apps
         abeona_2011_1.jar      - license protector
```

You can copy JAR files to any appropriate place for your development. It is not necessary to keep them in the installation directory.

2.1 Packages

JavaGantt consists of these packages:

- *eu.beesoft.gantt* - contains main classes and interfaces (JavaGantt, GanttModel, TimelineObject, Dependency)
- *eu.beesoft.gantt.action* - contains all implemented actions
- *eu.beesoft.gantt.chart* - contains chart related classes (ChartComponent, Layer, TimeSpan, TimeUnit)
- *eu.beesoft.gantt.treetable* - contains treetable related classes (GanttTreeTable, GanttColumn, GanttNode)
- *eu.beesoft.gantt.undo* - contains classes for undo / redo functionality (UndoStep, StateEditableObject)

JavaGantt strongly uses classes from our **Gaia** library. This is our company general library and is available for free. Current version of this library is included in your distribution, but you can upgrade at any time. These packages from *Gaia* are used by JavaGantt:

- *eu.beesoft.gaia.swing* - contains Swing components and classes for Swing support
- *eu.beesoft.gaia.util* - contains utility classes

2.2 License installation

JavaGantt is component protected by *Abeona protection system*. You can freely download this component and use it, but there is one feature built-in: if JavaGantt is not licensed, it stops to work 10 minutes after it was constructed. You must restart your application which uses this component.

When you purchase a license for JavaGantt, you will get a license file to your email. Copy this file where you want to your classpath and then complete your application with a similar code:

```
JavaGantt gantt = new JavaGantt ();

  ... init gantt component

  InputStream is = SomeMyClass.class.getInputStream (
        "path-to-license-file");
  License license = gantt.setLicense (is);
  if (license == null) {
       throw new RuntimeException (
        "Ooops, something goes wrong here...");
  }
```

3 Model for JavaGantt

The model for JavaGantt consists of:

- the *TimelineObject* interface
- the *Dependency* interface
- the *GanttModel* class

That's all. Your domain objects must implement the given interfaces. And your model must subclass the *GanttModel* class to serve some methods.

3.1 Programming domain objects

Every application works with data objects. These objects are important for the problem area that an application solves and they are called domain objects. For application, which uses gantt chart, are domain objects - *tasks*.

So the domain object class for our example is *Task*. What do you have to do to let it work with JavaGantt?

JavaGantt does not prescribe what objects you can use. But it requires to implement interface *eu.beesoft.gantt.TimelineObject*. It is a simple interface with methods like *getStartDate()*, *isMilestone()*, and so on. You can implement them very easy.

It is a bit complicated, if you use a *summary* tasks. This is the task which has subtasks - it is like container in the task hierarchy (non leaf node). Then you have to arrange synchronization of date changes in subtasks and in summary task.

In our demo you can see this code for class Task:

```
public class Task extends AbstractBean implements TimelineObject {

     private String taskId;
     private String name;
     private String description;
     private Date startDate;
     private Date endDate;
     private long effort;
     private boolean milestone;
     private Task supertask;
     private List<Task> subtask;
     private List<Dependency> dependencies;
     private Resource assignee;
```

```
public Task () {
	// empty
}


/** Standard getter for property 'name' */
public String getName () {
	return name;
}


/** Standard setter for property 'name' */
public void setName (String name) {
	String old = this.name;
	this.name = name;
	firePropertyChange ("name", old, name);
}



/**
 * Getter for property 'startDate'. Implementation of
 * TimelineObject interface.
 * If this task is summary, calculates start date from subtasks.
 */
public Date getStartDate () {
	if ((startDate == null) && isSummary ()) {
		updateStart ();
	}
	return startDate;
}


/**
 * Setter for property 'startDate'.
 * Implementation of TimelineObject interface.
 * If this task has a supertask, notifies it about the change –
 * invokes method updateStart().
 */
public void setStartDate (Date start) {
	if (Objects.equals (this.startDate, start)) {
		return;
	}
	Date old = this.startDate;
	this.startDate = start;
	if (supertask != null) {
		supertask.updateStart ();
	}
	firePropertyChange ("startDate", old, start);
}


/**
 * For summary task calculates its start date from subtasks.
 */
private void updateStart () {
	if (isSummary ()) {
		Date thisStart = null;
		List<Task> tasks = getSubtask ();
```

```
                    for (Task task : tasks) {
                            Date taskStart = task.getStartDate ();
                            if (taskStart != null) {
                                    if ((thisStart == null) ||
                                      thisStart.after (taskStart)) {
                                            thisStart = taskStart;
                                    }
                            }
                    }
                    if (thisStart != null) {
                            setStartDate (thisStart);
                    }
            }
    }

    ......

}
```

Remaining methods are implemented in the similar way. Of course, your implementation can be quite different. *TimelineObject* interface prescribes some methods, not their body.

Maybe you want to know, what is *AbstractBean*, which is used as parent class of our *Task*. It is class from our **Gaia** library (it is shipped with JavaGantt) and it serves as support for property change listeners and their notification.

3.2 Interface TimelineObject

TimelineObject is an interface that prescribes some methods for your domain objects. Each of these methods concerns on getting / setting data for the gantt chart:

- **void addPropertyChangeListener (PropertyChangeListener)** - Adds instance of java.beans.PropertyChangeListener to a listener list.
- **Date getStartDate ()** - Returns the start date of this TimelineObject.
- **void setStartDate (Date)** - Sets the new start date for this TimelineObject.
- **Date getEndDate ()** - Returns the end date of this TimelineObject.
- **void setEndDate (Date)** - Sets the new end date for this TimelineObject.
- **boolean isSummary ()** - Returns true if this is a summary object (it contains some subtasks).
- **boolean isMilestone ()** - Returns true if this is a milestone object.
- **void setMilestone (boolean)** - Sets the milestone property.
- **List<Dependency> getDependencies ()** - Returns a list of all dependencies between this TimelineObject and other objects.
- **void setDependencies (List<Dependency>)** - Sets a list of all dependencies between this TimelineObject and other objects.

A typical application's domain object that has to implement this interface is a *task*.

3.3 Interface Dependency

Dependency is an interface that describes dependency between TimelineObjects. Here is the list of methods you need to implement:

- **TimelineObject getSuccessor ()** - Returns a dependant object.
- **void setSuccessor (TimelineObject)** - Sets a dependant object.
- **TimelineObject getPredecessor ()** - Returns an object which dependant depends on.
- **void setPredecessor (TimelineObject)** - Sets object which dependant depends on.
- **int getType ()** - Returns the type of this Dependency (one of the constants FINISH_TO_START, FINISH_TO_FINISH, START_TO_START, START_TO_FINISH).
- **void setType (int)** - Sets the new type of this Dependency (one of the constants FINISH_TO_START, FINISH_TO_FINISH, START_TO_START, START_TO_FINISH).

There are some constants defined in the *Dependency* interface. They define the type of dependency:

- **FINISH_TO_START** - Constant for dependency type where dependant cannot begin until the object it depends on is complete.
- **FINISH_TO_FINISH** - Constant for dependency type where dependant cannot be completed until the object that it depends on is completed.
- **START_TO_START** - Constant for dependency type where dependant cannot begin until the object that it depends on begins.
- **START_TO_FINISH** - Constant for dependency type where dependant cannot be completed until the object that it depends on begins.

3.4 Class GanttModel

GanttModel extends JTreeTable model (*eu.beesoft.gaia.swing.TreeTableModel*) from Gaia library. You have to implement these methods:

- **TimelineObject createObject (TimelineObject, int, UndoStep)** - Creates a new domain object and adds it as child to parent (you don't need to care about treetable nodes). Register all modified objects to undo before any change.

- **boolean deleteObject (TimelineObject, UndoStep)** - Removes given object from its parent on domain objects level (you don't need to care about treetable nodes). Register object, its parent and all modified objects to undo before any change.
- **boolean moveObject (TimelineObject, TimelineObject, int, UndoStep)** - Moves given object on domain objects level (you don't need to care about treetable nodes). Register all modified objects to undo before any change.

The methods above are designed to manipulate with objects on the domain objects level. You don't need to work with treetable nodes.

Here is the implementation of some methods from our demo application:

```
    @Override
    public boolean moveObject (TimelineObject object,
            TimelineObject newContainer, int newIndex, UndoStep undo)
    {
            Task task = (Task) object;
            Task supertask = (Task) newContainer;
            undo.registerObject (task);
            undo.registerObject (supertask);
            Task oldSupertask = task.getSupertask ();
            if (supertask != oldSupertask) {
                    if (oldSupertask != null) {
                            undo.registerObject (oldSupertask);
                            oldSupertask.removeSubtask (task);
                    }
                    supertask.addSubtask (task);
            }
            supertask.shiftSubtask (task, newIndex);
            return true;
    }


    @Override
    public boolean deleteObject (TimelineObject what, UndoStep undo){
            Task task = (Task) what;
            Task parent = task.getSupertask ();
            undo.registerObject (parent);
            undo.registerObject (task);
            parent.removeSubtask (task);
            return true;
    }


    @Override
    public TimelineObject createObject (TimelineObject parent,
    int index, UndoStep undo) {
            undo.registerObject (parent);
            Task parentTask = (Task) parent;
            Task task = new Task ();
            String id = Gantt.getNextTaskId ();
            task.setTaskId (id);
            task.setName ("Task " + id);
            parentTask.addSubtask (task);
            parentTask.shiftSubtask (task, index);
            return task;
    }
```

Note, there is an *UndoStep* object used as argument in these methods. It supports undo / redo operations.

If you want to paint in the chart component on the time lines some texts (if you use LabelLayer or TooltipLayer), you need override these methods in GanttModel subclass:

- **String getLabel (TimelineObject)** - Returns label text for given object. This text uses LabelLayer to paint additional info to timeline object. Typically, for a task is name of the assignee probably the best result.
- **String getTooltipText (TimelineObject)** - Returns tooltip text for given object.

GanttModel supports **lazy loading** for treetable data. The possibilities of ExplorableTreeNode, which is an ancestor of GanttNode, is emplyed here. When node is expanded and was not explored still, method explore(GanttNode) is invoked.

- **void explore (GanttNode)** - This method is called by GanttNode for data loading.

Override this method to loading children of your domain object. In the typical lazy-loading implementation:

- get a user object from given node
- load children of user object and build hierarchy
- for each loaded child create instance of GanttNode and set child as its user object
- add each created instance of GanttNode as a child to the given node

4 JavaGantt component

JavaGantt is the Swing component which interacts with GanttModel. It covers these components:

- GanttTreeTable - displays treetable on the left side of gantt chart
- ChartHeader - displays header (months, dates) for the right side of the gantt chart
- ChartComponent - displays gantt drawing area (timeline objects and dependencies)

These components are constructed in the initialization phase and developer has no impact to it. You can change their properties only.

4.1 Getting standard sub-components

The standard JavaGantt subcomponents are accessible by these methods:

- **GanttTreeTable getTreeTable ()** - Returns the tree table object of this JavaGantt.
- **ChartComponent getChartComponent ()** - Returns the instance of ChartComponent.
- **ChartHeader getChartHeader ()** - Returns the instance of ChartHeader.
- **UndoManager getUndoManager ()** - Returns the undo manager for JavaGantt operations.
- **getDividerLocation ()** - Returns divider location of the split pane (between treetable and chart component).
- **setDividerLocation (int)** - Sets the divider location of the split pane (between treetable and chart component).

4.2 Building a treetable

Here are JavaGantt methods you can use to build a chart:

- **GanttModel getModel ()** - Returns GanttModel instance for this JavaGantt object.
- **void setModel (GanttModel)** - Sets the new model for JavaGantt object.
- **void addColumn (GanttColumn)** - Appends a column to the end of table columns.

- **GanttColumn addColumn (String, String)** - Appends a column to the end of table columns.
- **void removeColumn (GanttColumn)** - Removes the column from the treetable.
- **void removeColumn (String)** - Removes a column with given key from the treetable.

The methods working with columns are implemented in GanttTreeTable and methods in JavaGantt simply redirect to them.

4.3 Building a chart

Here are JavaGantt methods you can use to build a chart:

- **void addLayer (Layer)** - Adds given layer to the list of chart component layers.
- **void addLayer (Layer, int)** - Adds given layer to the list of chart component layers.
- **void removeLayer (Layer)** - Removes given layer from the list of chart component layers.
- **List<Layer> getLayers ()** - Returns the list of chart component layers.

All of these methods are implemented in ChartComponent and methods in JavaGantt simply redirect to them.

4.4 Mouse support

JavaGantt has built-in mouse support, so you don't need to work with mouse listeners. Use these methods instead:

- **void doubleClicked (GanttNode)** - Invoked when user double clicked on the node. In this implementation does nothing. Typically, you should display properties dialog for node's user object. In our demo application we construct and display task properties form in this method.
- **List<Action> getPopupActions (List<GanttNode>)** - Returns a list of actions for selected nodes. Invoked when user right clicks on the node(s) in treetable. In this implementation does nothing and returns null. If you will override it and return a list of actions, JavaGantt constructs popup menu and displays it.

Method *doubleClicked()* is invoked from treetable and also from gantt chart component. The *getPopupActions()* is treetable method only. If you return a non-null list, the popup menu is displayed.

4.5 Localization

The JavaGantt components (treetable columns and actions) intense use resource bundle technique. These methods of JavaGantt are about resource bundles:

- **String getResourceBundleName ()** - Returns the name of the resource bundle.
- **void setResourceBundleName (String)** - Sets the name of resource bundle. This name must according to java.util.ResourceBundle specification. Resource bundle is then used to construct treetable columns and gantt actions.

But resource bundle may be accessed also if you don't set its name explicitly. So how does it work?

The main class for these purposes is *eu.beesoft.gaia.util.Language*. In the Swing environment is a singleton (but it has a public constructor, so you can create as many instances as you need in server environment). The singleton is accessible via method

```
Language.getInstance ();
```

which returns last created instance of *Language*. Using *Language* has big advantage when you change application's Locale. Setting this into *Language* provides for all *LanguageListener*s notification to easy reloading texts from resource bundles.

When you take a look at Language API documentation, you can see method *getText()* with first argument either a resource bundle name or some object. These methods are used from JavaGantt and its subcomponents.

1. if resource bundle name is set (by method *JavaGantt.setResourceBundleName()*), this resource bundle is constructed and used. Note, that resource bundle name must have a form of class name.
2. if **no** resource bundle name is set, then the name of current JavaGantt class (its subclass) is taken and used to find resource. If resource is not found, or key with given name is not found, *Language* class takes a superclass name and repeats searching.

So if you don't use method *JavaGantt.setResourceBundleName()*, you **must** create a properties file **in the same package** where your JavaGantt subclass is stored and **with the same name** as thas subclass has. In that case it will work without explicitly setting the resource bundle name and you can use the advantages of inheritance.

4.6 License protection

JavaGantt is protected by the *Abeona* protection system.

It works without any license, but 10 minutes after JavaGantt was created it displays a warning message and stops to paint itself. So if you want to develop an application, which will work more than 10 minutes, you need purchase the license from BeeSoft. License is plain properties file protected by digital signature and here are methods to tell JavaGantt about license:

- **License setLicense (InputStream)** - Loads license from given licenseStream. This is a key method of library protection system.
- **License getLicense ()** - Returns instance of valid License or null.

The first method has an input stream from license file as argument, and returns newly created *License* instance. You don't need to work with this object, just check for non-null return value. If it is null, you have some problem with that file.

5 Gantt Tree Table

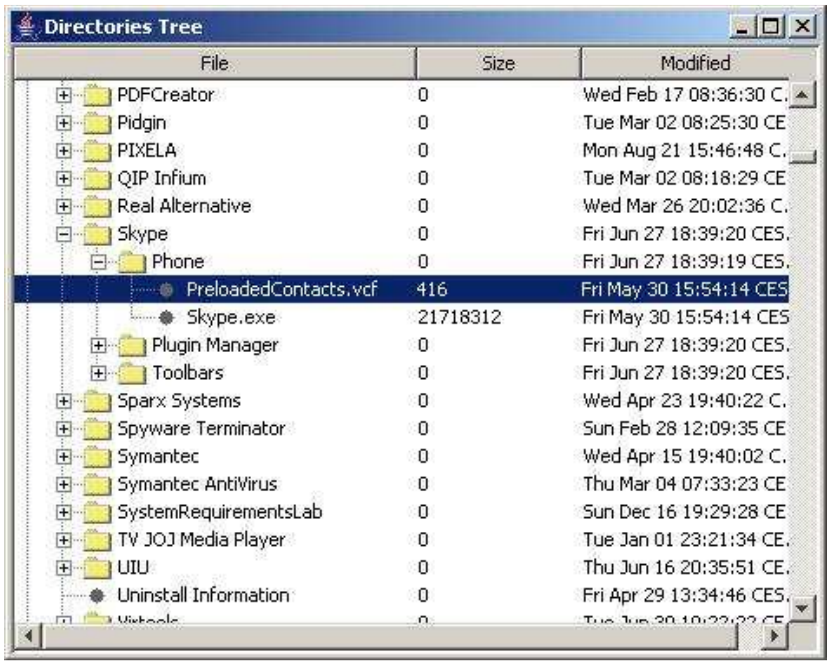
GanttTreeTable is the tree table for gantt chart. It is the left-located component of the JavaGantt.

GanttTreeTable is the descendant of the *eu.beesoft.gaia.swing.JTreeTable* (you can read about it in the next chapter, and the next informations are accessible in API documentation).

5.1 JTreeTable description

The *eu.beesoft.gaia.swing.JTreeTable* class is a combination of a JTree and a JTable - a component capable of both expanding and contracting rows, as well as showing multiple columns of data.

This component always displays a tree in the first column and the additional data in the next columns. You can see its usage in a very simple example of the directory tree.



This component has the next features:

- it inherits from its parent class ability to hide / show columns
- right click on the table header shows popup menu with list of all columns - this is the user way to show / hide columns
- also inherits possible "*zebra*" look (even rows can have an alternate background color)
- the tree is always painted in the first column of the treetable

- as its model you must use *eu.beesoft.gaia.swing.TreeTableModel* or subclasse it (that's what GanttModel does)
- descendants of the *eu.beesoft.gaia.swing.TreeTableNode* are only nodes acceptable in the JTreeTable (and GanttNode is the descendant of TreeTableNode)

5.2 GanttNode

GanttNode is the only *javax.swing.tree.TreeNode* acceptable by a gantt tree data structure. It is a final class, so there is no way to customize it.

Here are the key features of the GanttNode:

- a TimelineObject is used as an user object in tree node. Each user object must implement this interface.
- it inherits ability of the data lazy-loading
- although GanttNode is a part of the treetable, it holds some informations for chart component (the bounds of timeline, for example)

5.3 Gantt column and data binding

GanttColumn represents a column in the GanttTreeTable. It is a descendant of *javax.swing.table.TableColumn* . It supports:

- resource bundle usage for column label
- data binding for getting / setting values from GanttNode's user object

Data binding is implemented by *eu.beesoft.gaia.util.Miner* class. It gets values from any object and sets them in. In the first step via property accessors (getter / setter) and if appropriate accessor is not found, on the field access level. More informations about *Miner* class you can find in API documentation.

GanttModel has implemented these methods related to data binding:

- **Miner getMiner ()** - Returns the current instance of the Miner used by this GanttModel.
- **void setMiner (Miner)** - Sets a new instance of the Miner for use by this GanttModel. The Miner class is designated to obtain property value on method or field level from any object. You can subclass Miner class to reach your requirements.
- **Object getValueAt (TreeTableNode, int)** - Returns value from given node and for given column.
- **void setValueAt (TreeTableNode, int, Object)** - Sets given value to the user object in the node and column.

All you need to do for automated data binding is to set property name to gantt column. Then methods *getValueAt()* and *setValueAt()* in GanttModel can work fully automatically. If your domain objects are not suitable for data binding, override methods *getValueAt()* and *setValueAt()*. You can also subclass Miner class and set it to GanttModel by method *setMiner(Miner)*.

These methods support data binding in **GanttColumn**:

- **String getBinding ()** - Returns binding for this column. Binding is a property name (or dot-separated chain of property names). A value for this column is mined from this property.
- **void setBinding (String)** - Sets binding for this column. Binding is a property name (or dot-separated chain of property names). A value for this column is mined from this property.

Data binding works very simply (from the programmer's point of view). It uses class *eu.beesoft.gaia.util.Miner* to get (and also set) the value from (to) object. Value is defined by the property or field name. *Miner* tries to find getter for property, and if this was not successful, it uses field access.

So all you need to do is to tell the column about property (field) to display:

```
column.setBinding ("startDate");
```

You can use dot-convention for property name, if your property is not in given object, but in referenced object. In our demo application has class *Task* the field *assignee*. So if you want to display a task assignee's name in column, you can set binding this way:

```
column.setBinding ("assignee.name");
```

5.4 Localization

A *GanttColumn* has built-in support for resource bundle usage. These methods are implemented:

- **String getResourceBundleKey ()** - Returns key used for resource bundle.
- **void setResourceBundleKey (String)** - Sets resource bundle key. This key is used when accessing resource bundle to get (localized) column label.

- **void languageChanged (Language)** - Implements LanguageListener. Invoked when environment changes. Reinitializes column header from resource bundle.

First two methods operates with resource bundle key. It is a key under which is stored localized text in resource bundle. The third method is a listener method and is invoked, when you change language settings in application. It is invoked from *eu.beesoft.gaia.util.Language*.

So you don't need to invoke method *column.setHeaderValue ()*. This is the usual way, known from JTable and its components. You can set a key for column name in resource bundle instead:

```
column.setResourceBundleKey ("columnStartDate");
```

and in your JavaGantt resource bundle to define something like:

```
columnStartDate=Start date
```

When you or user changes the application language environment, you have to take a care only of localized resource bundle.

6 Gantt Chart

Gantt chart is located on the right side of the JavaGantt component. It consists of:

- chart header
- chart component

All classes of the gantt chart are in the package *eu.beesoft.gantt.chart*.

There are two important classes for whole package:

- TimeUnit
- TimeSpan

TimeUnit is an enumeration for time unit granularity. It contains constants from seconds to year and some useful methods.

TimeSpan envelopes date range (start time - end time). Instances of *TimeSpan* are created by *ChartComponent* in its *updateChart()* method. They contains information about:

- start date of time span
- end date of time span
- whether it is a holiday or weekend
- x-position of this time span in chart component coordinates
- width of this time span in pixels

All of these informations are used by chart component to prepare painting.

The painting in gantt chart is implemented through the layers. A *Layer* is an abstract class that provides functionality to paint chart and interact with a user.

When you construct a gantt, you set to the ChartComponent several layers. Chart component, when painting, iterates through them and invokes the *paint(Graphics)* method on each of them. The order you add layers to ChartComponent is important. Each layer paints just the things it is programmed for. You can assemble the chart from the layers as you need. So the chart preparation can look like this:

```
JavaGantt gantt = ...
gantt.addLayer (new BackgroundLayer ());
gantt.addLayer (new CalendarLayer ());
gantt.addLayer (new GridLayer ());
gantt.addLayer (new TodayLayer ());
gantt.addLayer (new GanttNodeLayer ());
gantt.addLayer (new LabelLayer ());
gantt.addLayer (new DependencyLayer ());
```

All these layers are available in the JavaGantt distribution.

But you are not limited to use just these layers. You can write your own, or modify existing, or to omit some of them, to get the original look for your application.

6.1 Chart header

ChartHeader paints headers above the ChartComponent instance. This component contains two headers - major and minor - and their models.

HeaderModel is used to paint major or minor header for ChartComponent. It contains data for header columns.

HeaderColumn is a subclass of *javax.swing.table.TableColumn* to hold list of *TimeSpan* instances. These instances are used to calculate sizes of the column.

There is very low probability you need to customize these classes. They are created by JavaGantt in initialization phase. All you need to do is to use them.

6.2 Working with the time scale

The Gantt chart time scale is significant for gantt chart span, for the chart header content and also for chart clarity and readability.

All classes of the gantt chart header and the time scale are in the package *eu.beesoft.gantt.chart*:

- **TimeUnit** is an enumeration for time unit granularity. It contains constants from seconds to year and some useful methods.
- **TimeSpan** envelopes date range (start time - end time). Instances of *TimeSpan* are created by *ChartComponent* in its *updateChart()* method. They contain information about start and end date of time span, whether it is a holiday or weekend, x-position of this time span in chart component coordinates and width of this time span in pixels.
- **ChartHeader** paints headers above the ChartComponent instance. This component contains two headers - major and minor - and their models.
- **HeaderModel** is used to paint major or minor header for ChartComponent. It contains data for header columns.
- **HeaderColumn** is a subclass of *javax.swing.table.TableColumn* to hold list of *TimeSpan* instances. These instances are used to calculate sizes of the column.

There is very low probability you need to customize these classes. They are created by JavaGantt in initialization phase. All you need to do is to use them.

Zoom policy

The zoom policy defines the way the chart is zoomed in / out. There is defined an interface *ZoomPolicy* and also its default implementation built in ChartComponent. But maybe you will need create your own.

The *ZoomPolicy* requires to implement 4 methods:

- public int getStepCount () - returns the step count for this policy
- public TimeUnit getMajorStep (int index) - returns time granularity for major header
- public TimeUnit getMinorStep (int index) - returns time granularity for minor header
- public int getTimeSpanWidth (int index) - returns width (in pixels) of TimeSpan instances for given step.

They could be implemented in this manner, for example:

```
public class MyZoom implements ZoomPolicy {

     private TimeUnit[] majorStep;
     private TimeUnit[] minorStep;
     private int[] widthStep;

     public Zoom () {
            majorStep = new TimeUnit[5];
            minorStep = new TimeUnit[5];
            widthStep = new int[5];

            majorStep[0] = TimeUnit.WEEK;
            minorStep[0] = TimeUnit.DAY;
            widthStep[0] = 80;

            majorStep[1] = TimeUnit.MONTH;
            minorStep[1] = TimeUnit.DAY;
            widthStep[1] = 40;

            majorStep[2] = TimeUnit.MONTH;
            minorStep[2] = TimeUnit.WEEK;
            widthStep[2] = 20;

            majorStep[3] = TimeUnit.YEAR;
            minorStep[3] = TimeUnit.WEEK;
            widthStep[3] = 10;

            majorStep[4] = TimeUnit.YEAR;
            minorStep[4] = TimeUnit.MONTH;
            widthStep[4] = 5;
     }


     public int getStepCount () {
            return majorStep.length;
     }
```

```
    public TimeUnit getMajorStep (int index) {
           return majorStep[index];
    }


    public TimeUnit getMinorStep (int index) {
           return minorStep[index];
    }


    public int getTimeSpanWidth (int index) {
           return widthStep[index];
    }
}
```

How to customize text for chart header

When *HeaderModel* (re)constructs itself, it calls method *GanttModel.getChartHeaderText (Date, TimeUnit, boolean)* for each relevant date. So this method in *GanttModel* is the place to customize chart header texts.

What you have to do is to return text for given Date and time granularity (TimeUnit). For example, if TimeUnit is YEAR, you will return "2010" if date is of that year. The third (boolean) argument distinguishes between request for major and minor header, because you may want to display the same value differently in each other header.

6.3 Chart component

A *ChartComponent* is one of the most meaningful sub-components of JavaGantt. It is painted on the right side of JavaGantt (there is an instance of JTreeTable on the left).

It co-ordinates painting with registered instances of the Layer class and dispatches events to them.

The ChartComponent is created by JavaGantt in initialization phase and there is no way to customize it.

6.4 Layer

A layer is an object having a graphical representation that can be displayed in the gantt chart and that can interact with the user.

The *Layer* class is the abstract superclass of the each gantt chart layer.

The gantt *ChartComponent* holds a stack of user-defined layers. When the method *paint(Graphics)* on ChartComponent is invoked, it walks through this stack (in order the layers were added) and if the layer is visible (method *isVisible()* returns true) calls method *paint(Graphics)* on it.

Here are the methods that *eu.beesoft.gantt.chart.Layer* class provides:

- **JavaGantt getGantt ()** - Returns instance of JavaGantt to which this Layer belongs.
- **boolean isVisible ()** - Returns value of property 'visible'.
- **void setVisible (boolean)** - Sets a new value for property 'visible'. Layer is painted only if visible = true.
- **List<GanttNode> getPaintedNodes ()** - Returns list of currently on-screen visible and painted gantt nodes.
- **int getPosition (Date)** - Returns x-coordinate for given date.
- **void mouseClicked (MouseEvent)** - Invoked when the mouse has been clicked on a component. In this implementation does nothing.
- **void mouseEntered (MouseEvent)** - Invoked when the mouse enters a component. In this implementation does nothing.
- **void mouseExited (MouseEvent)** - Invoked when the mouse exits a component. In this implementation does nothing.
- **void mousePressed (MouseEvent)** - Invoked when a mouse button has been pressed on a component. In this implementation does nothing.
- **void mouseReleased (MouseEvent)** - Invoked when a mouse button has been released on a component. In this implementation does nothing.
- **void mouseMoved (MouseEvent)** - Invoked when the mouse button has been moved on a component (with no buttons no down). In this implementation does nothing.
- **void mouseDragged (MouseEvent)** - Invoked when a mouse button is pressed on a component and then dragged. In this implementation does nothing.

There is a set of implemented layers you can use. Of course, you can write your own - subclass *eu.beesoft.gantt.chart.Layer* and implement method *paint(Graphics)*.

You can also modify an existing layer - subclass it and override method you need.

The good news is that you don't need to take a care of adding objects to layers and their synchronization with JavaGantt model. Layers work directly with JavaGantt model nodes. A *GanttTreeNode* holds its *bounds* - it is a rectangle where this node is painted in JavaGantt ChartComponent (Layer) coordinates. This can your work greatly facilitate.

Now we can see how to customize some layer. We have the *GanttNodeLayer* which paints timeline objects (tasks, summaries and milestones). And now we want to paint the tasks differently. We want to paint task's time line with colorized information about task completion.

The easiest way to implement this is to subclass *GanttNodeLayer* and override method *paintTask()*:

```
public class MyLayer extends GanttNodeLayer {

     @Override
     protected void paintTask (GanttNode node, Graphics g) {
             Rectangle bounds = node.getBounds ();
             int x = bounds.x;
             int y = bounds.y;
             int width = bounds.width;
             int height = bounds.height;

             // don't fill whole space for node, it looks uglily
             if (width > 6) {
                     x += 2;
                     width -= 5;
             }
             y = y + height / 4 - 1;
             height = height / 2 - 1;

             // paint border around task time line
             g.setColor (getTaskBorderColor ());
             g.drawRect (x, y, width, height);

             // edit coordinates
             ++x;
             ++y;
             --width;
             --height;

             // get the task completion and compute threshold
             Task task = (Task) node.getUserObject ();
             double completion = task.getCompletion();
                 // value between 0.0 and 100.0
             int threshold = (int) (width * completion) / 100;

             // paint completed part of task with darker color
             if (threshold > 0) {
                     g.setColor (getTaskInteriorColor ().darker());
                     g.fillRect (x, y, threshold, height);
             }

             // paint incomplete part of task with brighter color
             if (threshold < 100) {
                     g.setColor (getTaskInteriorColor ().brighter());
                     g.fillRect (x + threshold, y, width - threshold,
                                 height);
             }
     }
}
```

6.5 Implemented layers

Here is the list of the implemented layers you can use:

- **BackgroundLayer** - Paints colored background for gantt chart.
- **CalendarLayer** - Paints background for weekend days.
- **DependencyLayer** - Paints dependencies between gantt nodes.
- **GanttNodeLayer** - Paints gantt nodes as time-line objects on chart component and processes moving and dragging of nodes by mouse.
- **GridLayer** - Paints grid (horizontal and vertical lines) in gantt chart component.
- **LabelLayer** - Paints text label beside node.
- **TodayLayer** - Paints strong vertical line on chart component at given date (usually today).

7 Gantt actions

As a standard Swing component, JavaGantt is controlled by actions.

7.1 AbstractGanttAction

AbstractGanttAction is a base class for gantt component actions. It is not necessary to subclass this class, you can use any action. But AbstractGanttAction offers these features:

- ability to return JavaGantt instance
- built-in resource bundle and language support
- undo / redo support

AbstractGanttAction processes *actionPerformed (ActionEvent)* method to offer undo / redo support. It invokes *executeAction (ActionEvent, UndoStep)* method, which is an abstract method, and if it returns true, registers all changes from UndoStep in UndoManager.

7.2 Localization

AbstractGanttAction gets its properties from resource bundle. All action properties (currently: text, icon and tooltip) must be described in resource bundle. This process is automated, so you must observed some rules:

- each property name must be prefixed by fully qualified or simple action class name
- property name for action text (label) is **text**
- property name for action icon is **icon** - it is path to icon image on classpath
- property name for action tooltip text is **tooltip**

For example, if your action class is *mydomain.mypackage.MyAction*, then in the resource bundle should be:

```
mydomain.mypackage.MyAction.text=Text for action
mydomain.mypackage.MyAction.icon=mydomain/mypackage/MyIcon.gif
mydomain.mypackage.MyAction.tooltip=This is a tooltip for my action
```

or shorter:

```
MyAction.text=Text for action
MyAction.icon=mydomain/mypackage/MyIcon.gif
MyAction.tooltip=This is a tooltip for my action
```

7.3 Undo / Redo

JavaGantt uses Swing *undo* package functionality to provide support for undo/redo operations.

There are two classes in *eu.beesoft.gantt.undo* package for that purpose:

- UndoStep
- StateEditableObject

StateEditableObject implements *javax.swing.undo.StateEditable* interface. Its constructor takes one object, and StateEditableObject can this object introspect and to remember changed properties.

UndoStep is the descendant of *javax.swing.undo.CompoundEdit* class. It serves as a container of all operations for one undo / redo step. It offers two methods:

- **void registerObject (Object)** - Registers object to store / restore its state. If given object does not implement interface javax.swing.undo.StateEditable, the object is covered by StateEditableObject to make it eligibly for undo / redo operations. Then is object registered and its pre-state is obtained.
- **void end ()** - Gets the post-edit state of the required objects and ends the edit.

If you want to process undo / redo, you have to:

1. create an instance *UndoStep*
2. call *registerObject(Object)* for each object that can be changed in executed action **before** any change occurs
3. call *end()* after action finished
4. obtain from JavaGantt instance an UndoManager
5. on UndoManager instance invoke *addEdit (UndoStep)* method

AbstractGanttAction supports all of these steps except step 2. So what you have to do, if your action is deriver from *AbstractGanttAction*, is to call *registerObject(Object)* for each object that can be changed in method *executeAction (ActionEvent, UndoStep)* only.

7.4 Implemented actions

Here is the list of the implemented actions you can use:

- **CollapseAllNodesAction** - This action collapses all nodes in the JavaGantt treetable.
- **CreateNodeAction** - This action asks *GanttModel.createObject(TimelineObject, int, UndoStep)* to create new domain object and then creates a new GanttNode} and includes it in hierarchy.
- **DeleteNodeAction** - Action asks *GanttModel.deleteObject(TimelineObject, UndoStep)* to delete domain object and if it is successful, removes the node from tree hierarchy.
- **ExpandAllNodesAction** - This action expands all nodes in the JavaGantt treetable.
- **IndentNodeAction** - This action asks *GanttModel.moveObject(TimelineObject, TimelineObject, int, UndoStep)* to move a domain object into domain hierarchy and then moves also corresponding GanttNode.
- **MoveNodeDownAction** - This action asks *GanttModel.moveObject(TimelineObject, TimelineObject, int, UndoStep)* to move a domain object into domain hierarchy and then moves also corresponding GanttNode.
- **MoveNodeUpAction** - This action asks *GanttModel.moveObject(TimelineObject, TimelineObject, int, UndoStep)* to move a domain object into domain hierarchy and then moves also corresponding GanttNode.
- **RedoAction** - This action asks *JavaGantt.getUndoManager()* to redo next operation.
- **TodayAction** - This action scrolls the date/time columns in the ChartComponent} so the column with today date is visible. You can set any date for this action as *today* date.
- **UndoAction** - This action asks *JavaGantt.getUndoManager()* to undo last operation.
- **UnindentNodeAction** - This action asks *GanttModel.moveObject(TimelineObject, TimelineObject, int, UndoStep)* to move a domain object into domain hierarchy and then moves also corresponding GanttNode.
- **ZoomInAction** - This action zooms in the content of ChartComponent.
- **ZoomOutAction** - This action zooms out the content of ChartComponent.

8 Shipping to customer

When you will deploy your application based on the JavaGantt at your customer (or you will ship it), your product must contain libraries from our JavaGantt *lib* directory:

- **javagantt_2011_1.jar**
- **gaia_2011_2.jar**
- **abeona_2011_1.jar**

Also must include the product license stored in JavaGantt root directory:

- **license.txt**

This text file must be accessible and readable by the user of your product.

Your distribution also must include the license protection file you obtain when you purchase JavaGantt.

- **javagantt.license**

You should store this file to your classpath (e.g. into some of your JAR files).

No other files are required.